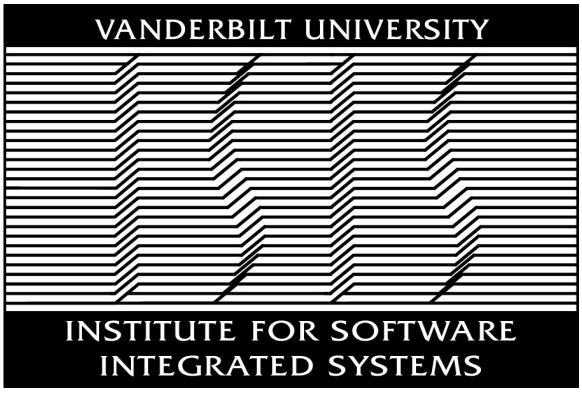

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, 37212

Deliberative Reasoning in Software Health Management

Nagabhushan Mahadevan, Abhishek Dubey , Daniel Balasubramanian,
Gabor Karsai

TECHNICAL REPORT

ISIS-13-101

April, 2013

Deliberative Reasoning in Software Health Management

Nagabhushan Mahadevan, Abhishek Dubey, Daniel Balasubramanian, and Gabor Karsai

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

Abstract. Rising software complexity in aerospace systems makes them very difficult to analyze and prepare for all possible fault scenarios at design-time. Therefore, classical run-time fault-tolerance techniques, such as self-checking pairs and triple modular redundancy are used. However, several recent incidents have made it clear that existing software fault tolerance techniques alone are not sufficient. To improve system dependability, simpler, yet formally specified and verified run-time monitoring, diagnosis, and fault mitigation are needed. Such architectures are already in use for managing the health of vehicles and systems. Software health management is the application of adapting and applying these techniques to software. In this paper, we briefly describe the software health management technique and architecture developed by our research group. The foundation of the architecture is a real-time component framework (built upon ARINC-653 platform services) that defines a model of computation for software components. Dedicated architectural elements: the Component Level Health Manager (CLHM) and System Level Health Manager (SLHM) are providing health management services: anomaly detection, fault source isolation, and fault mitigation. The SLHM includes a diagnosis engine that uses a Timed Failure Propagation (TFPG) model derived from the component assembly model, and it reasons about cascading fault effects in the system and isolates the fault source component(s). Thereafter, the appropriate system level mitigation action is taken. The main focus of this article is the description of the fault mitigation architecture that uses goal-based deliberative reasoning to determine the best mitigation actions for recovering the system from the identified failure mode.

1 Introduction

Software has become the key enabler for a number of core capabilities and services in modern systems. It is also increasingly used for system integration [30]. For example, a modern car contains about 20 million lines of embedded code, while just the flight controls avionics of modern aircraft (e.g. F-22, F-35) contains 1.7-5.7 million lines of code [9]. The scale of these systems imposes many challenges to ensuring correct and proper behavior, especially in avionics where software malfunctions have caused a number of incidents in the past, including but not limited to those referred to in these reports: [32,3,4,20]. In [41], Sha provides an excellent discussion on the complexity in avionics software.

Safety critical software systems, while in operation, must be able to adapt to and mitigate the effects of latent faults in their implementation, in software, in hardware, or in the larger system, even if those faults appear simultaneously. State of the art techniques for safety critical systems involve applying software fault tolerance principles, methods and tools to ensure that a system can survive software defects that manifest themselves at run-time [27,26,46,8,36].

However, several incidents mentioned above indicate the inadequacy of these techniques and point to the need for additional approaches that apply anomaly detection, fault source identification (i.e. diagnosis), fault effect mitigation, and fault prognosis, as defined and used in System Health Management of complex engineering systems [34,22]. One such approach has been termed Software Health Management. It is a run-time technique and it includes fault detection, isolation, and mitigation activities to remove fault effects [44]. Recent work in this area includes [35,40,28,5].

We have developed an architecture and supporting model-based tools for implementing software health management functions for component-based systems. The foundation of the architecture is a real-time component framework (built upon an ARINC-653 platform) that defines a specific model of computation for software components [13]. This framework uses the concepts of temporal isolation, spatial isolation, and strict temporal deadlines from ARINC-653 and combines them with the well-defined component interaction patterns derived from the CORBA Component Model [48]. Health management in the framework is performed at two levels: the Component Level Health Manager (CLHM) provides localized and limited service

for managing the health of individual components while a System Level Health Manager (SLHM) manages the health of the overall system.

SLHM employs a diagnosis engine based on a Timed Failure Propagation Graph (TFPG)[1]. The TFPG model is automatically synthesized from the model of components and their connectivity; the engine reasons about fault-effect cascades in the system and isolates the fault source components. This is possible because the data and behavioral dependencies (and hence the fault propagation) across the assembly of software components can be deduced from the well-defined and restricted set of interaction patterns supported by the framework [14]. In the past, we showed how system-wide mitigation can be performed based on reactive timed state machines specified by the designer at system integration time [28]. However, one of the problems with this approach to fault mitigation at a system level is the complexity of the specifications required to cover all possible combinations of failure scenarios.

This paper describes an approach to system level fault mitigation using deliberative, goal-oriented reasoning to identify alternate component configurations that can restore the desired system functionality. We build upon our earlier work [16] to provide design-time support for the concise specification of functional goals, the redundancy available to support these goals, and component-specific operational requirements. We describe the runtime framework and its use of off-the-shelf constraint solvers to search for alternate configurations that can restore system functionality. The specific case of using off-the-shelf Boolean Satisifiability (SAT) solvers is discussed in detail, along with results from illustrative examples and a larger case study. We also outline the steps for integrating pseudo-Boolean solvers into the runtime framework.

The outline of this paper is as follows. Section 2 presents background material from our earlier work in the context of Software Health Management. Section 3 motivates the need to move from a prescriptive to a deliberative, search-based reasoning. The design-time and run-time support for the deliberative strategy are discussed in Sections 4 and 5, respectively. Section 6 focuses on the representation of the problem to use Boolean Satisfiability (SAT) solvers and showcases the results using a specific SAT solver with small illustrative examples and a larger case study. Section 7 outlines the encoding required for integrating pseudo-Boolean solvers. Section 8 contains a discussion of the work. Section 9 presents related work and Section 10 concludes.

2 Background: Software Health Management

System level health management and fault tolerance approaches are often rely on the notion of interacting components. Hence, it is natural to apply these concept of health management to systems built from software components, where each software component is developed and tested individually, and then monitored and managed at run-time. In our work, the first step was to develop and implement such a component model.

2.1 A real-time component framework

The ARINC-653 component model (ACM) is built upon the services of ARINC-653: an industry standard for safety critical operating systems [2]. ARINC-653 systems group *processes*[1] into spatially and temporally separated *partitions*, with one or more partitions assigned to each *module* (i.e. a processor), and one or more modules forming a *system*.

Spatial partitioning ensures exclusive use of a memory region by an ARINC partition. It also guarantees that a faulty process in a partition cannot ruin the data structures of other processes in other partitions, isolating, for instance, low-criticality vehicle management software components from safety-critical flight control software components. Temporal partitioning ensures exclusive use of processing resources by a partition. A fixed periodic schedule is used by the real-time operating system (RTOS) to share the resources between partitions. This deterministic scheduling ensures that each partition is allowed exclusive access to the processor within its predetermined execution interval. It also guarantees that when the predetermined execution interval of a partition is over, the partition's execution will be interrupted, the partition will be placed into a dormant state and the next partition in the schedule order will be granted exclusive access to the processor.

[1] An ARINC-653 process is a unit of concurrency that is analogous to a thread in a desktop operating system such as Linux.

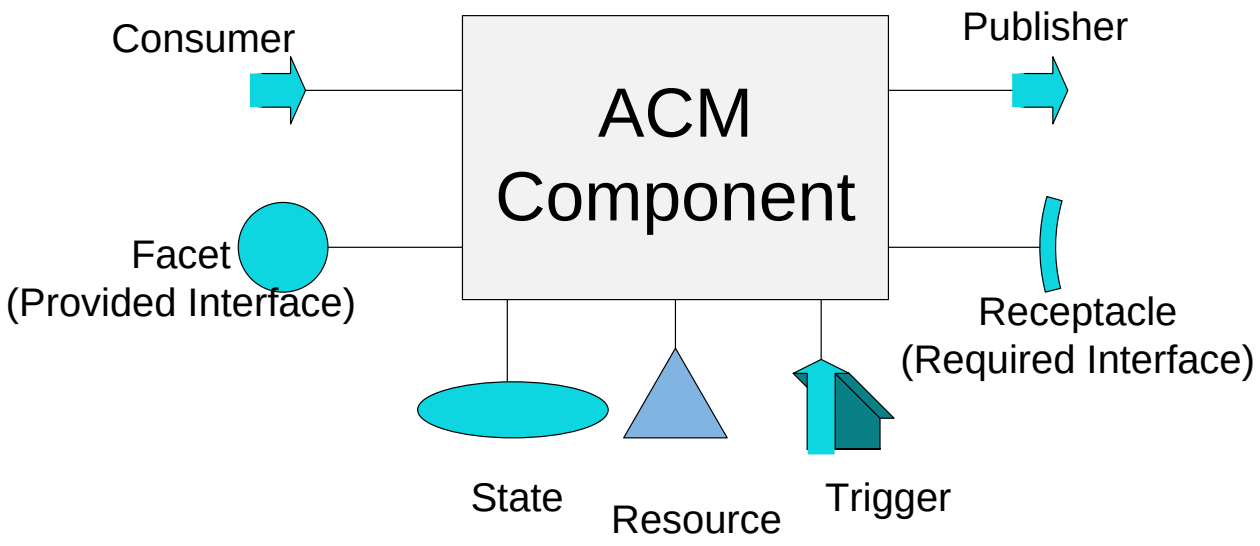

Fig. 1: ACM Component

The ARINC-653 component model allows developers to group a number of ARINC-653 processes into a reusable component. A component is a group of processes that share state but do not interact directly. However, components do interact with each other via well-defined interaction patterns (chosen from a fixed set), facilitated by ports. In ACM, a component can have four kinds of external ports for interactions: *publishers*, *consumers*, *facets* (provided interfaces[2]), and *receptacles* (required interfaces), as shown on Figure 1.

Each port has an interface type: a named collection of methods, for provided and required ports, or an event type: a data structure, for publishers and subscribers. The component can interact with other components through **synchronous** call/return interfaces (associated with providers and required ports), and/or **asynchronous** publish/subscribe event connections (assigned to publisher and consumer ports). Additionally, a component can host internal methods that are periodically triggered. Most of these interactions borrow concepts from other software component frameworks, most notably from the CORBA Component Model (CCM) [48].

The component model also provides guidance on the allocation of activities to a component. Because this framework is designed for hard real-time systems, it is required that each port is statically allocated to an ARINC-653 process. In the case of a facet port, the interface (supported by the facet) could include more than one method. The facet port is then assigned a dedicated ARINC-653 process to handle all methods of the interface. Furthermore, the access to component state is synchronized by a component-wide lock, allowing at most one ARINC-653 process per component to be active. In other words, a component is always single-threaded. Please see [13] for a detailed description.

The framework implementing the ARINC-653 component model consists of two parts: a Linux-based runtime environment (the includes a ARINC-653 emulator), and a modeling environment with associated design tools. Together these tools allow systems to be developed in two distinct phases. The first phase is performed by the component developer. A component is a reusable artifact that provides one or more functions that can be developed and verified independently, and stored in a repository for reuse. Often, component developers organize various components into subsystems. The second phase is completed by the system integrator. The system integration includes modeling and configuring the system architecture, as well as deploying the components on computing hosts. Both phases are assisted by a suite of model-driven tools. [3].

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

[3] The modeling environment and the Linux runtime are available for download from `https://wiki.isis.vanderbilt.edu/mbshm/index.php/Main_Page`

Component Execution States: A component can be in one of the following three states: *active*, *inactive* and *semi-active*. When a component is in the inactive state, none of the component ports (i.e. processes) are operational. In the other two states, the component is fully (active) or partially (semi-active) operational. In the active state, all the component ports perform their task. In the semi-active state, only the consumer and required ports of a component are operational, the publisher and provider ports are not. While a component is executing, i.e., it is in the active or semi-active state, the operations of component ports can introduce faults in the system, which can lead to anomalies in either the same component or in a connected component.

Example: Figure 2 shows the assembly for a notional GPS system with a redundant set of Sensor/GPS component pairs: Sensor plus GPS, and Sensor2 plus GPS2. Here, each sensor component (i.e. Sensor and Sensor2) publishes an event every 4 sec which is consumed by the associated GPS component (i.e. GPS and GPS2) at that rate. Thereafter, each GPS component publishes an event, which is sporadically consumed by the Navigation Display (NavDisplay) component. The Navigation Display component fetches location data from the GPS and GPS2 components by using the provider port called 'gps'.

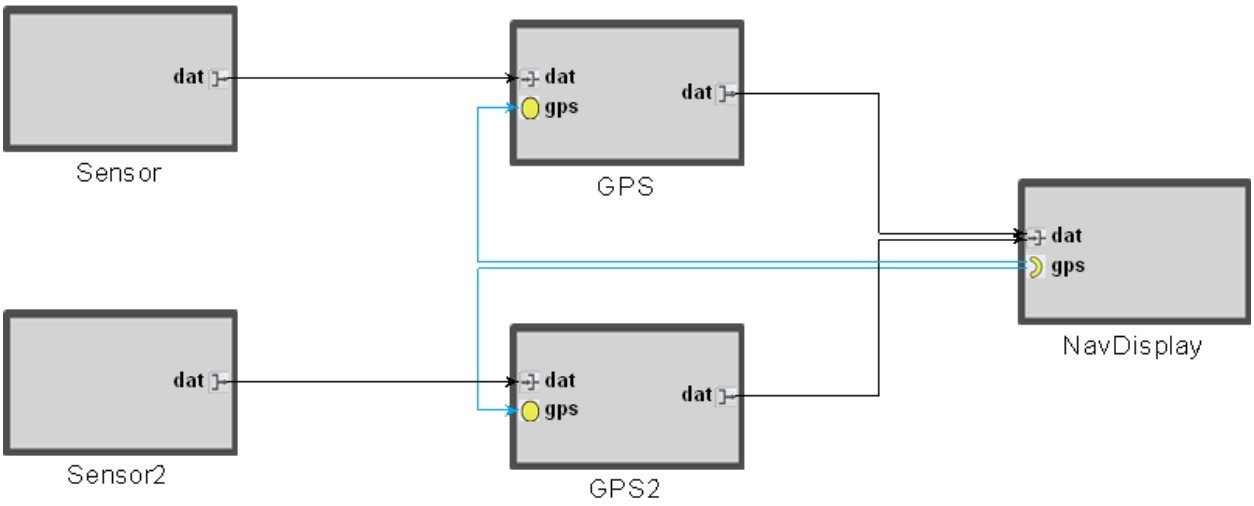

Fig. 2: GPS Software Assembly

In the initial setup of the assembly, the Sensor, GPS, and NavDisplay components are in the *active* state, allowing the Navigation Display to receive updates from the active Sensor and GPS components. The redundant Sensor2 and GPS2 components are set to stand-by mode, ready to replace the functioning Sensor and GPS components in the case of a problem. Sensor2 is set to the active state, and GPS2 is set to the semi-active state, allowing GPS2 to update its state by collecting data through its active consumer port from the Sensor2 component. Being in semi-active state, GPS2's publisher and provider ports do not service the NavDisplay component.

2.2 Failure scenarios and anomaly monitoring

We consider two primary failure sources during the operation of each component port: (a) a concurrency fault, and (b) a latent bug in the source-code associated with the port. The concurrency fault is caused when the port is unable to obtaining the lock associated with the component, leading to delayed or lack of execution of the port. On the other hand, the latent-bug in the source-code could lead to an incorrect execution in the component port. Both of the above fault sources can lead to several secondary anomalies in either the same component or in a connected component. In the ACM framework, the design tools allow the system designer to deploy monitors, which can be configured to detect deviations from expected behavior

as well as violations to specifications expected in ports or components. The following discrepancies can be currently identified using these monitors:

- *Lock timeout*: The framework implicitly generates monitors that check for starvation. Each component has a lock (to avoid interference among callers), and if a process does not obtain the lock within a specified time, an anomaly is declared. The value of the timeout is either set to a default value equal to the deadline of the process associated with the component port or can be specified by the system designer.
- *Data validity violation* (only applicable to consumers): Any data token consumed by a consumer port has an associated expiration age. This is also known as the validity period in ARINC-653 sampling ports. We have extended this to be applicable to all types of component consumer ports, both periodic and aperiodic.
- *Pre-condition violation*: Developers can specify conditions that should be checked before executing. These conditions can be expressed over the current value or the historical change in the value, or rate of change of values of variables (with respect to the previously known values for same parameter), such as
  1. the message in asynchronous calls,
  2. the function parameters of synchronous calls, and
  3. the (monitored) state variables of the component.
- *User code failure*: Any error or exception raised in the user code can be abstracted by the software developer as an error condition which can then be reported to the framework. Any unreported error is recognized as a potentially unobservable discrepancy.
- *Post-condition violation*: Similar to pre-condition violations, but these conditions are checked after the execution of the operation associated with the component port.
- *Deadline violation*: Any process execution must finish within the specified deadline.

These monitors can be specified via (1) attributes of model elements (e.g. deadline, data validity, lock time out), and (2) via a simple expression language (e.g. conditions). While deadline, data validity and lock timeout are defined as relative timeouts expressed in seconds, the conditions (both pre-conditions and post-conditions) are written as logical expressions using the conventional logical and comparison operations, over (1) current value (of an argument, say x), (2) delta value (change in value since the last sample, written as $delta(x)$), or(3) rate value (rate of change, written as $rate(x)$).

Code-generators included in the design tools produce the appropriate (C++) code for the monitors. All monitors, other than those observing deadline violations, are evaluated in the same thread as executing the component port. The monitors detecting deadline violations are run on framework threads, so that they can observe the CPU resource usage of the concerned port while that is executing. A violation detected by any of the monitors is considered as an anomaly and is reported to the health management system.

The components in the assembly model (Figure 2) have been instrumented with monitors to detect anomalies related to resource usage (detected as deadline violations), user code (detected as user code violations), age of the received data (detected as data validity violations), as well as violations on the contracts (pre-conditions and post-conditions) for exchanging data between the ports [14,**?**]. They also have the capability to mitigate any problems that could arise because of the anomaly, but this mitigation action may not remove the primary source of failure. Realizing the benefits and limitations of each strategy, we implemented a two-level health management strategy in our framework: the *component level* that is local to a component, and the *system level* that includes the entire assembly of components. While the component level health manager (CLHM) is specified by the component developers, the system level health manager (SLHM) is provided by the system integrator.

2.3 Local mitigation

The Component Level Health Manager(CLHM) provides localized and limited functionality for managing the health of a component. The CLHM is modeled as a timed-state machine that can be customized for each component. It is triggered by anomalies detected by the monitors (as described in the previous section) deployed inside the component and reacts with the appropriate local mitigation action [14]. In addition to these monitors that detect and report anomalies, monitors to report the starting and stopping of a port's process can also be selected. These monitors aid in building *observers* to track the activation sequences of

component processes (ports) and report any deviations from the expected sequence. Observers are modeled as parallel state machines within the CLHM, with one machine acting as an observer and another as the health manager. Each of the parallel state machines could be triggered by their relevant monitor events. While the observer tracks the state evolution, the health manager takes appropriate mitigation actions for the anomalies detected. When an anomaly is detected in the observer, it triggers the health manager portion of the CLHM state machine and that takes the appropriate mitigation action.

A detailed discussion on the local-mitigation actions of the CLHM as well as example CLHM models customized to react to violations in user code, deadlines, pre-conditions, post-conditions and data validity in the Sensor, GPS and NavDisplay components are presented in [14,**?**]. The anomalies observed and the mitigation actions taken by the CLHM are local to the component. The CLHM provides a 'quick fix' and could be effective in handling temporary local faults as well as arresting the fault propagation. A system wide mitigation engine would be ill-suited to react to every local anomaly.

2.4 Diagnosis and system-level mitigation

In component-based systems, anomalies in a component can be local, or they can be the result of a secondary effect caused by an anomaly in an upstream component. Identifying this pattern is important in order to isolate the root failure source. While the component level mitigation code (provided by the component developer) can quickly react to the local anomaly, this does not guarantee that the primary source of failure is mitigated. A system-wide mitigation engine would be better suited to identify and mitigate the real fault source, especially when the failure effects cascade across component boundaries. The SLHM in ACM relies on a Timed Failure Propagation Graph (TFPG) model [1] model that captures the failure modes (fault sources), discrepancies (anomalies) and the failure propagation among them, across the entire system. The TFPG model for the complete component assembly can be automatically generated based on the knowledge of the components (their ports), the data and control flow dependencies between the ports of a component (i.e., intra-component) and the inter-component interactions (captured in the assembly model). A detailed discussion of the TFPG model templates for the component ports, as well as the failure propagation patterns between ports (publisher and consumer, provided and required ports) in the context of assembly models similar to Figure 2 is discussed in [14,**?**].

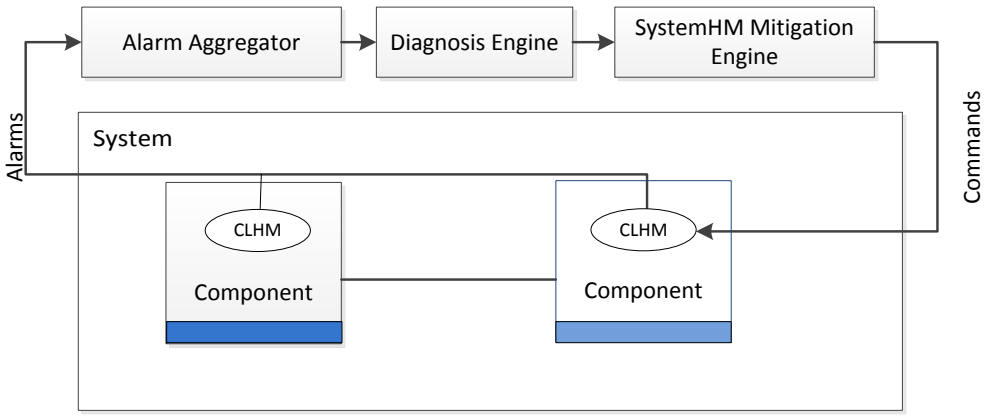

Fig. 3: SLHM architecture

The System Level Health Management (SLHM) engine uses the information from the local CLHMs (anomalies observed and mitigation actions taken) in the context of the TFPG models to locate the possible fault source (component) and identify the possible set of mitigation actions that could restore the health of the system. The execution of the SLHM requires augmentation of the ACM assembly with three special SLHM components: *Alarm Aggregator*, *Diagnosis Engine*, and *SystemHM Mitigation Engine*, as shown in Figure 3. These components are described briefly in the following paragraphs.

The *Alarm Aggregator* is responsible for collecting and aggregating inputs from the component level health managers (local alarms and the corresponding mitigation actions). This information is processed using a moving window with a length of two hyperperiods of the complete partition schedule. The events are sorted based on their time of occurrence and then supplied to the diagnosis engine.

The *Diagnosis Engine* contains an instance of a Timed Failure Propagation Graph reasoning engine. The reasoner uses the TFPG model (auto-generated for the given component assembly) to isolate the most plausible fault source: a component whose fault could explain the observations i.e. anomalies detected and the CLHM commands issued. The diagnosis result, which is a list of one or more faulty components, is then reported to the *SystemHM Mitigation Engine*, which computes the mitigation action.

The *SystemHM Mitigation Engine* receives the diagnosis results (the set of faulty components) and responds with an appropriate, system level command to mitigate the fault and its effects.

The system level mitigation described in [28,**?**] uses a reactive mitigation technique that employs a timed parallel state machine. The models captured the mitigation actions for each failure scenario in the examples: a GPS System [15] and an Inertial Measurement Unit of an avionics suite [28]. The state machine models refer to the component states (component execution modes) and are triggered by the fault diagnosis report from the diagnosis engine. When the appropriate guard conditions (that are based on the fault state of one or more components) are satisfied, the mitigation actions (reconfiguration commands) are generated as part of the state transition. The new state (after the transition) reflects the component state (execution mode) after reconfiguration.

While the mitigation strategy based on the reactive timed state machine was effective, the prescriptive approach to system level mitigation proved extremely cumbersome. In the following sections, we lay the foundations and describe in detail our recent work on an alternate mitigation scheme based on a deliberative search strategy to restore the system functionality.

3 Shifting from a reactive to a deliberative mitigation strategy

The reactive mitigation strategy used a prescriptive model in which the mitigation action for each failure scenario in each component configuration needed to be modeled in a timed state machine model. While the use of hierarchical and parallel state machines helped reduce the complexity of the models, this prescriptive approach was still very tedious, cumbersome and error-prone. The rest of this paper explores an alternate deliberative mitigation strategy and our results on applying this to the System Level Health Management.

The SLHM mitigation strategy based on a *Deliberative Mitigation Engine* is similar to the one with a *Reactive Mitigation Engine* in that it receives the diagnosis results (the set of faulty components) and responds with an appropriate set of system level commands to mitigate the fault and its effects. However, the similarity ends there. Unlike the reactive engine in which the mitigation action for every failure scenario needs to be prescribed (in our case with a timed state machine model), the deliberative engine relies on models of system goals and functionalities and function- llocation models that indicate the specific groups of components that provide the desired services. The deliberative engine identifies the functionalities affected by the faulty components and attempts to restore the failed (or degraded) functions by searching the function allocation models for alternate component configurations.

The paradigm shift implied by this deliberative mitigation strategy required additional modeling and runtime support in the ACM framework. Design-time additions include support for models that capture the system goals, the redundancy available to support the desired functionality, and any component-specific operational requirements. At runtime, the SLHM layer should support a generic framework that can formulate the problem in a way that allows a constraint solver to search through the configuration space and identify alternate configurations to restore the functionality. The following sections describe in detail our approach to supporting such design and runtime frameworks that provide a mitigation engine based on deliberative reasoning.

4 Design time support: Modeling system functions and functional redundancy

This section details the extensions made to the ACM modeling framework to support a deliberative, search-based mitigation strategy. The extensions include support for modeling: (1) system functions as a functional

decomposition tree, (2) redundancy available to support those functions in terms of function allocation models, and (3) operational requirements for each component. In addition, this section also includes definitions of new properties and their semantics (mathematical relations) that serve as the basis for formalizing the search problem used in the deliberative mitigation strategy.

4.1 Modeling system goals

At design time, the system designer enumerates the functional requirements and goals of a system in the form of a tree structure. Each tree represents a top-level function that the system must support in a specific mode of operation. The tree structure follows the functional decomposition of the system, where intermediate nodes denote sub-functions that are all required in order to support a higher-level function, while the leave nodes are primitive functions that cannot be decomposed further.

Example: Figure 4 shows the functional requirement tree model for the GPS assembly shown on Figure 2. The figure shows the top-level function of the GPS assembly which is to determine the position of the vehicle in inertial space (`Inertial Position`), represented by the root. For this `Inertial Position` function to be active both its children nodes: `Body Acceleration Measurement` and `GPS-Position` must be available. In a typical vehicle the continuous tracking of the inertial position depends on the measurement of the acceleration of the concerned body (`Body Acceleration Measurement`) and a continuously updating filter (e.g., a Kalman Filter) that constantly computes and updates the body's estimated position. For this filter to work correctly it has to be regularly updated with high accuracy position data(`GPS-Position`).

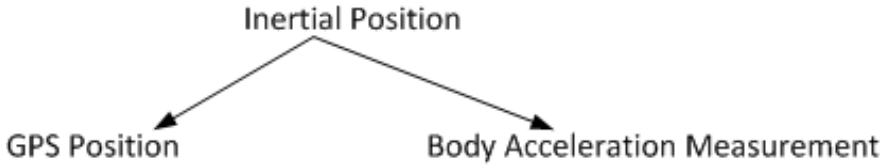

Fig. 4: Example of system functions for the GPS Assembly in Figure 2.

Semantics : The semantics of the functional requirement tree can be formally expressed in terms of an *isActive* boolean attribute that is defined for each function. The *isActive* attribute for a function captures whether the specific functionality is actively provided by the system. Equation 1 captures the formal relationship between the *isActive* property of the functions in a functional requirement tree.

$$isActive(f_p) = \bigwedge_{\forall f \in \mathbb{F}} isActive(f) \tag{1}$$

where

f_p is the root function and

$\mathbb{F}$ is the set of all child functions

4.2 Modeling the function allocation

The function allocation model captures the logical group of components that can support a specific function or goal. The set of components related to a function can be hierarchically composed into the groups of the following types:

1. ALT Group: **Exactly 1** out of N components is required to support a function X. This is expressed as $X \rightarrow \texttt{EXACTLY}(1, C1, C2, ...CN)$
   where $C1, C2, ...CN$ are the N components.

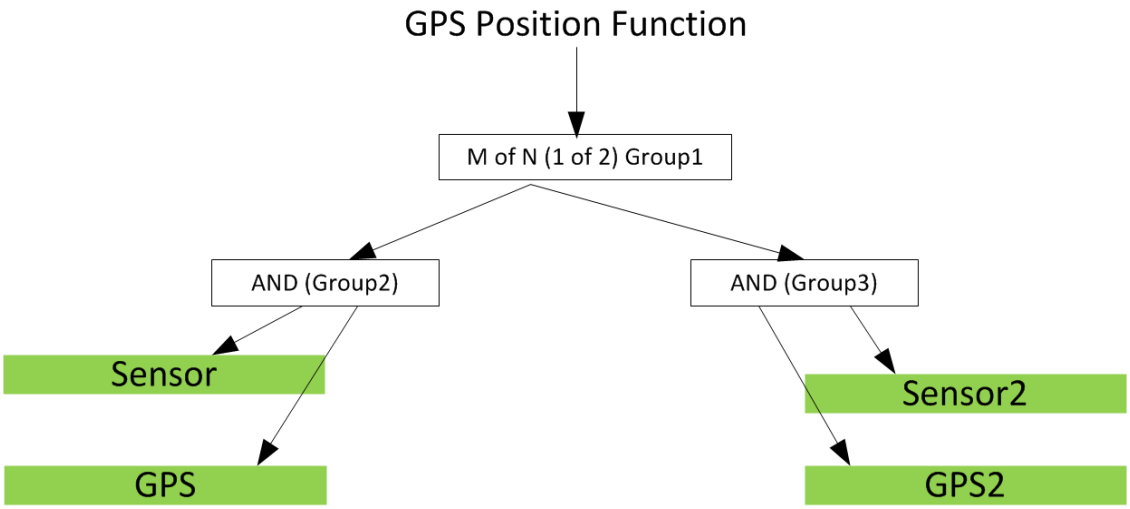

Fig. 5: Function allocation model for GPS-Position function in Figure 4

2. M-of-N group: **At least** M out of N components are required to support a function X. This is expressed as $X \rightarrow \texttt{ATLEAST}(M)(C1, C2, ...CN)$
3. AND Group: **All** components are required to support a function X. This is expressed as $X \rightarrow \texttt{ALL}(C1, C2, ...CN)$.

Once specified, the function allocation tree contains: (1) a system function as its root node, (2) groups as intermediate nodes, and (3) software components as the leaf nodes. Note that the ALT and M-of-N groups capture the redundancy available to provide the desired functionality.

Example: Figure 5 shows the function allocation model for one of the functions in Figure 4 using the components in the assembly depicted in Figure 2. The model indicates that providing the GPS Position function requires at least one of the two Groups: Group1 and Group2. This is represented by an MofN (1 of 2) Group. Further, it indicates that Group1 and Group2 are both AND groups. Hence, Group1 requires the services of all its child nodes: the Sensor and the GPS component. Likewise, Group2 requires the services of the Sensor2 and GPS2 components.

Semantics: The formal relationship between the nodes (function, group, component) in the function allocation model are captured in terms of two boolean attributes: $isUsable$ and $isActive$ of each node of the tree. The $isActive$ attribute represents if the node is currently active. The $isUsable$ attribute represents whether the node is usable, i.e., it can provide the desired service. In the case of a software component, the $isActive$ attribute is reflective of the component's execution state in the ACM framework (see 2.1). A component is active ($isActive = true$) when it is executing in an active mode in the ACM framework. The fault status of a component determines its usability. A component that is not faulty is regarded as usable ($isUsable = true$).

The $isActive$ attribute of a function and a group is determined by the $isActive$ property of their child nodes. A function is considered active ($isActive = true$) when all of its child nodes in the function-allocation model are active. An AND group is active if and only if all of its children are active. An ALT group is usable if exactly one of its child is active. An MofN group is active if at least M children are active. Exactly the same set of rules apply for determining the $isUsable$ property of a group and a function. These rules for determining the $isActive$ and $isUsable$ attributes are summarized in Tables 1 and 2, respectively. Note that in these Tables (1 and 2), g means group and c means component. Operator $child(x)$ returns the set of immediate children of x, and $|.|$ is the cardinality operator.

Table 1: $IsUsable$ semantics

Type	Definition
Component	$isUsable(c) \Leftrightarrow \neg isFaulty(c)$
And-Group	$isUsable(g) \Leftrightarrow (\forall x \in child(g))(isUsable(x))$
ALT-Group	$isUsable(g) \Leftrightarrow (\exists x \in child(g))(isUsable(x))$
MofN-Group	$isUsable(g) \Leftrightarrow (\exists X \subseteq child(g))(\lvert X \rvert \geq M)$ $(\forall x \in X)(isUsable(x))$
Function	$isUsable(f) \Leftrightarrow (\forall x \in child(f))(isUsable(x))$

4.3 Component Operational Requirement (COR) model

The function allocation model described in the previous section relates a system function to one or more groups of components that can provide that function. As the component assembly models show, the interactions between a component's ports (publisher and consumer ports and requires and provides ports) establish an inherent interdependency between these components. Based on these component interdependencies, a function allocation model should exhaustively enumerate all the components required to support it. However, this can quickly turn into an error-prone and cumbersome task, duplicating the information already contained in the assembly model. In order to keep the function allocation model concise and to avoid duplication of information, the ACM modeling language supports the specification of a Component Operational Requirement models for components.

For a component to be operational, the external dependencies of that component should be satisfied. In other words, its consumer and requires ports need the services of corresponding publisher and provider ports, respectively. In an assembly model a consumer port can be connected to multiple publishers and a requires port can be connected to multiple provides ports.

The component operational requirements of a component are captured in terms of its local ports. A *publisher* is active if its parent component is active. A *provider* is active if its parent component is active. A *consumer* is expected to actively receive and process data from only one of the its suppliers: a *publisher* port. Therefore, a *consumer* port, c , is operational when the parent component of exactly one of the supplier *publisher* ports connected to c is active[4]. In ACM, each *requires* port is expected to receive and process data from only one of its service providers: a *provides* port. Thus, a *requires* port, r, is operational when the parent component of at least one of its service provider's *provider* ports connected to r is active.

Implicit and explicit COR Model

While the implicit component operational requirement condition assumes that all the consumer and requires ports in a component need to be serviced for a component to be operational, this may not always be the criteria for some components to be operational. A component may not need all of the consumer and requires ports to be functional. A component may be considered completely operational if only certain groups of consumer and requires ports are functional. Based on the expected behavior of a component, the designer might be able to identify alternate groups of consumer and requires ports (within the component) that would keep the component operational.

Extensions to the ACM modeling language allow the designer to explicitly model any component's operational requirement. This model is similar to the function allocation model in that it allows a hierarchical

[4] Note: In ACM, a component does not have multiple publishers of the same kind. Hence, at most one publisher of a component services a specific consumer port of another component

Table 2: *isActive* semantics

Type	Definition
Component	$isActive(c)$ is marked by the deployment scheme and any previous action of the reasoner
And-Group	$isActive(g) \Leftrightarrow (\forall x \in child(g))(isActive(x))$
ALT-Group	$isActive(g) \Leftrightarrow$ $(\exists x \in child(g))(isActive(x))$ $(\forall y \in child(g))$ $(y \neq x)(\neg isActive(y))$
MofN-Group	$isActive(g) \Leftrightarrow$ $(\exists X \subseteq child(g))(\lvert X \rvert \geq M)$ $(\forall x \in X)(isUsable(x))$ $(\forall y \in child(f)/X)(\neg isUsable(y))$
Function	$isActive(f) \Leftrightarrow (\forall x \in child(f))(isActive(x))$

composition of AND/ALT/MofN groups. However, unlike the function allocation model, the members of the groups in a component operational model include the consumer and requires ports of the component.

Example: Implicit COR Model
As stated before, the implicit component operational requirement assumes that all the consumer and requires ports of a component need to be serviced for the component to be operational. In the context of the GPS components in the assembly model shown in Figure 2, this would mean that for the GPS component to be operational, its consumer port should be serviced. Because the consumer port is serviced by a single publisher from the Sensor component, the implicit component operational requirement for the GPS component is that the Sensor component be active. The same reasoning applies for the Implicit Component operational requirement for the GPS2 component, which depends on an active Sensor2 component. For the NavDisplay component, the Implicit Component Operational requirement translates into the consumer and requires ports both being active. Based on the discussions earlier, the consumer port requires `EXACTLY`(1,GPS, GPS2) and the requires port needs `ATLEAST`(1)(GPS,GP2) components to be active. Because the Sensor and Sensor2 components do not have any consumer or requires port, they do not depend on any other component in order to be operational.

Example: Explicit COR Model
To illustrate the explicit specification of a component operational requirement, the original NavDisplay component (Figure 2) is slightly modified, as shown in Figure 6. The modified NavDisplay component has two requires ports (gps1, gps2) as opposed to only one requires port (gps) in the original. The assembly model is also suitably modified. In the original assembly model (Figure 2), the single `requires` port in NavDisplay is serviced by two providers: one in component GPS and the other in component GPS2. In the modified model (Figure 6), the requires port gps1 is serviced by a provider port in the component GPS, and the requires port gps2 is serviced by a provided port in component GPS2.

Additionally, Figure 7 illustrates an explicit component operational requirement for the modified NavDisplay component. It states that the modified NavDisplay component needs `EXACTLY`(1,gps1, gps2) of the requires ports to be serviced. Given the interconnections in the assembly model in Figure 6, this implies that the modified NavDisplay component requires `EXACTLY`(1,GPS, GPS2) components. The original NavDisplay component in Figure 2 relies on implicitly derived component operational requirements using Equation 4.

Semantics: Table 3 and Equation 2 help to formalize the semantics associated with the component operational requirement model. Table 3 expresses the isActive property of a consumer port, c, and requires port, r, in

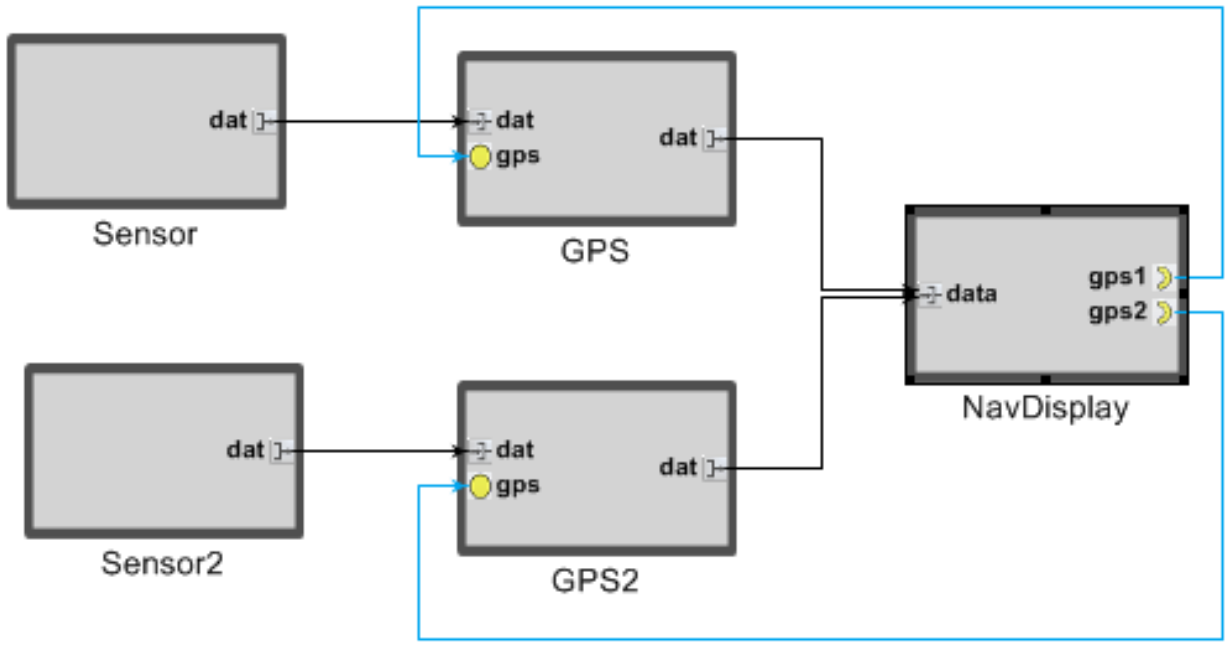

Fig. 6: Assembly model with modified NavDisplay.

Table 3: *isActive* for consumer and requires ports

Type	Definition
Consumer,c	$isActive(c) = ALT(isActive(P_{pub}))$ where P_{pub} is the set of Parent components of all Publisher ports connected to Consumer port,c
Requires,r	$isActive(r) = MofN_{M=1}(isActive(P_{pro}))$ where P_{pro} is the set of Parent components of all Provider ports connected to Requires port, r

terms of the `isActive` property of the parent components that service these ports. The `isActive` property of the consumer port is related to the `isActive` property of the parent components, P_{pub}, of the publisher ports that service the consumer port(c). The `isActive` property of the requires port, r, is expressed in terms of the `isActive` property of the parent components, P_{pro}, of the provider ports that service the requires port (r).

$$COR(comp) = f(gc_1, gc_2, ..., gc_n, gr_1, gr_2, ..., gr_m) \tag{2}$$

where

$$gc_i = isActive(c_i)$$

$$gr_i = isActive(r_i)$$

$comp$ is the Component

c_i refers to the ith consumer port in $comp$

r_i refers to the ith requires port in $comp$

Equation 2 defines the Component Operational Requirement (COR) criteria for a component $comp$ in terms of the `isActive` properties of the consumer and requires ports in $comp$. Using the relations defined in Table 3, it can be inferred that the COR for a component $comp$, depends on the `isActive` property of

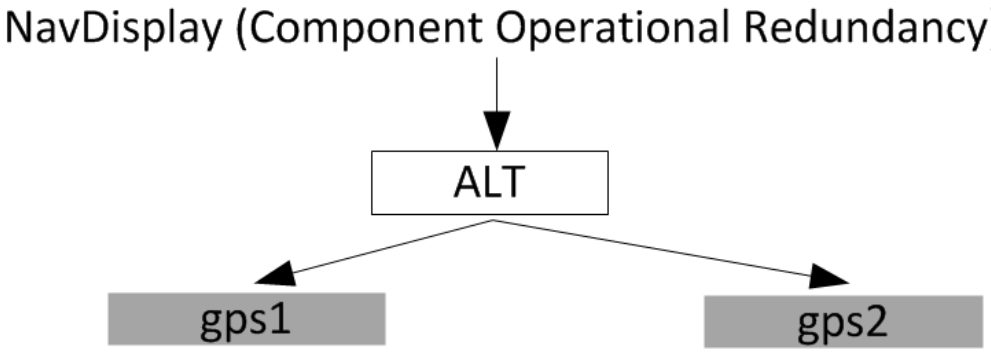

Fig. 7: Explicit Component Operational Requirement model in modified Display component.

the components containing the publisher and provider ports that service the consumer and requires ports in *comp*.

$$isActive(comp) \implies COR(comp) \tag{3}$$
where *comp* is a Component

Furthermore, the `isActive` property for any component, *comp*, is implicitly related to the *COR* criteria for the component (Equation 3). When the `isActive` property of a component is set to true, it implies that the component needs to be operational, or its *COR* property has to be true, which in turn captures the dependency on the `isActive` property of the components servicing its consumer and requires ports. This dependency chain over the `isActive` property of components in an assembly model can be used to effectively prune the requirements captured in the function allocation model.

While the explicit component operational requirement caters to the generic format of *COR* captured in Equation 2, the Implicit Component Requirement has a more specific form captured in Equation4. Apart from capturing the main aspect of the implicit *COR*, i.e., that all (AND-Group) of the consumers and requires ports in a component be actively serviced, Equation 4 also captures the redundancy available to serve the consumers and requires ports in a component. The exact expression for the `isActive` property of each consumer and requires port in a component can be derived using the equations in Table 3 and the component interaction and dependency information captured in the assembly model. These can then be substituted into Equation 4 to derive the exact implicit component operational requirement expression for each component.

$$ICOR(comp) = \bigwedge_{c \in C} isActive(c) \wedge \bigwedge_{r \in R} isActive(r) \tag{4}$$
where
comp is the Component
R Set of all Requires port (r) in *comp*
C Set of all Consumer port (c) in *comp*

Example: Implicit COR Expression: The implicit component operational requirement expression for each component in the GPS Assembly (Figure 2) can be derived by using Equation 4 and the relations in Table 3. Because the Sensor and Sensor2 components do not have any consumer or requires ports, they do not have any implicit *COR* expressions. The implicit *COR* expression for the GPS and GPS2 components are expressed in Equation 5. On the basis of substituting these results in Equation 3, the isActive property of the GPS depends on *Sensor* and that of GPS2 depends on *Sensor2* (Equations 6, 7).

$$ICOR(GPS) = isActive(Sensor)$$
$$ICOR(GPS2) = isActive(Sensor2) \quad (5)$$

$$isActive(GPS) \implies ICOR(GPS) \text{ i.e.}$$
$$isActive(GPS) \implies isActive(Sensor) \quad (6)$$

$$isActive(GPS2) \implies ICOR(GPS2) \text{ i.e.}$$
$$isActive(GPS2) \implies isActive(Sensor2) \quad (7)$$

$$ICOR(NavDisplay)$$
$$= isActive(dat) \wedge isActive(gps) \quad (8)$$

$$isActive(dat)$$
$$= ALT(isActive(GPS), isActive(GPS2)) \quad (9)$$

$$isActive(gps)$$
$$= MofN_{M=1}(isActive(GPS), isActive(GPS2)) \quad (10)$$

$$ICOR(NavDisplay)$$
$$= ALT(isActive(GPS), isActive(GPS2)) \wedge$$
$$MofN_{M=1}(isActive(GPS), isActive(GPS2)) \quad (11)$$

$$ICOR(NavDisplay)$$
$$= ALT(isActive(GPS), isActive(GPS2)) \quad (12)$$

$$= ALT(isActive(GPS) \wedge isActive(Sensor),$$
$$isActive(GPS2) \wedge isActive(Sensor2)) \quad (13)$$

The Implicit Component Operational Requirement expression for the NavDisplay component is captured in Equation 8 in terms of the isActive properties of its consumer (*dat*) and requires (*gps*) ports. Further, based on the relations defined in Table 3, Equations 9 and 10 capture the isActive property for the consumer port (*dat*) and requires port (*gps*). Substituting these expressions for the isActive property of the consumer port (*dat*) and requires port (*gps*) into Equation 8 results in Equation 11. The implicit component operational requirement for the NavDisplay component further simplifies to Equation 12, as ALT is more restrictive than MofN with M=1, and both ALT and MofN are expressed over the same set of variables (`isActive(GPS)` and `isActive(GPS2)`).

Based on Equations 6 and 7, it is evident that whether GPS or GPS2 is active depends on Sensor and Sensor2, respectively. Incorporating this recursive dependence information into Equation 12 (which suggests that NavDisplay depends on either GPS or GPS2), yields Equation 13, suggesting that the *COR* of NavDisplay depends on either group (GPS and Sensor) or group (GPS2 and Sensor2) to be active.

Notice that the dependency captured in the exhaustive function allocation model (Figure 5) corresponds to the dependency relationship captured in the operational requirement of NavDisplay (Equation 13). Hence, the exhaustive function allocation model might be substituted with a more succinct one (Figure 8) when the component operational requirements are taken into consideration.

Example: Explicit COR Expression: The NavDisplay component in the assembly model in Figure 6 has an explicit component operational requirement model (Figure 7). Equation 14 captures the equivalent mathematical relationship for the explicit *COR* obtained by using the relations defined in Table 3.

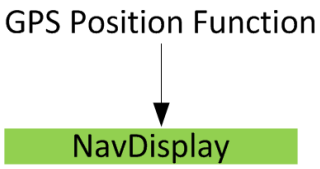

Fig. 8: Allocation model when the COR is considered for the GPS Position function shown in Figure 4 w.r.t. the assembly in Figure 3.

$$\begin{aligned} & COR(NavDisplay) \\ &= ALT(isActive(gps), isActive(gps2) \\ &= ALT(isActive(GPS), isActive(GPS2)) \\ &= ALT(isActive(GPS) \wedge isActive(Sensor), \\ & \quad isActive(GPS2) \wedge isActive(Sensor2)) \end{aligned} \tag{14}$$

The inclusion of the component operational requirement specification ensures that the function allocation model can focus on the core component(s) and the redundancy available for meeting the requirement. The additional dependency of these core components can be inferred from the assembly model by using the explicit component operational requirement model (if specified) or by deriving the implicit operational requirement constraint. The designer should exercise due care to ensure that the explicit or implicit operational constraints capture the core requirements and redundancies correctly. While the implicit COR constraints are derived only for those components where the explicit COR constraint is not specified, the designer, if required, can direct (through the model) the deliberative reasoner to ignore the COR constraints for specific components in the assembly.

This section explained in detail the modeling concepts introduced to support the deliberative search based mitigation strategy. While the model of system functions and goals helps define the services desired from the software system and capture any dependency between these functions, the function allocation model bridges the domain of expected functions with the alternate groups of service-providers: the components in the software assembly. Finally, the component operational requirements model states the dependencies between the various components (and the associated inherent redundancy present in the assembly model), thereby allowing for a concise specification in the function allocation model. The next section focuses on the runtime infrastructure that implements the deliberative, search-based mitigation strategy.

5 Run-time framework

The execution of a deliberative, search-based mitigation strategy requires a run-time framework represents the design-time artifacts as data structures that incorporate the inherent relationships between the various elements of the design space (functions, groups and components) and support algorithms to search for alternate configurations whenever the system functionality is affected. This section details such a run-time framework (Figure 9) deployed in the the SLHM layer of the ACM component assembly.

As outlined in Figure 9, the run-time framework includes the following activities:

1. Initialization of a run-time model representing the system goals, function allocation, component operational requirements, and the component assembly models from models created at design-time
2. Initialization of the run-time model based on the initial component configuration in the assembly model.
3. Actions of a Deliberative Reasoner (DR) undertaken after fault diagnosis is performed:
   - Initializing the run-time model based on the set of faulty components identified by the diagnosis engine.
   - Encoding the problem's search space as a constraint satisfaction problem.
   - Solving the constraint satisfaction problem to identify alternate component configurations that can restore the system functionality.
   - Updating the run-time model based on the new configuration.
   - Issuing the appropriate commands to reconfigure the system.

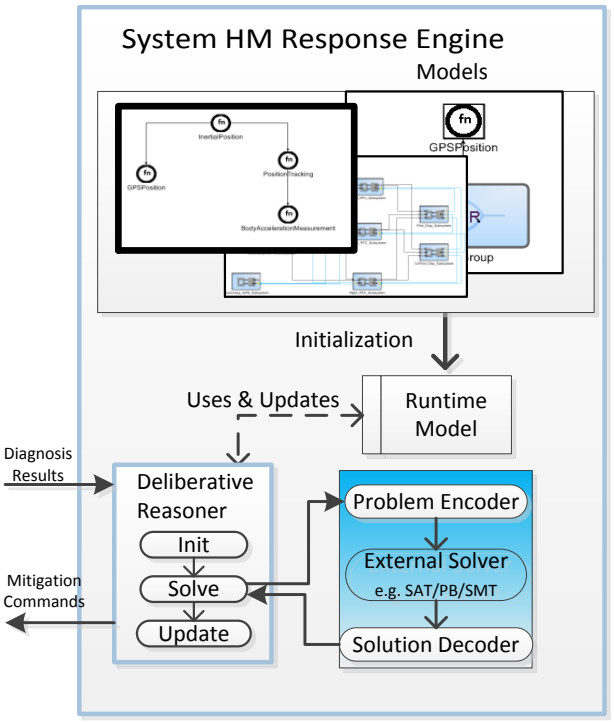

Fig. 9: Outline of the run-time framework to support deliberative, search-based strategy

5.1 Run-time model

The run-time system to support the deliberative search strategy depends on the information stored in the static models created at design time, including the software component assembly, the system goals, the function allocation, and the component operational requirements models. The resulting data structures also need to capture the relationship between the different elements of the models: the functions, the logical groups, and the software components. The run-time model is stored as a tree, similar to the dependency tree captured in the static models. The root node corresponds to the system's top-level functions, and the initial tree structure mimics the functional decomposition model. Thereafter, the tree is further expanded based on parsing each function allocation model in a top-down fashion. This allows the run-time model (tree) to grow and to incorporate nodes that correspond to the logical groups (AND/ ALT/ MofN) and finally include the nodes that correspond to the components in the assembly. The resulting tree may include multiple nodes that correspond to the same component and/or function. A separate map is maintained to capture the unique source (function or component) corresponding to each node. Figure 10 is representative of a portion of the tree that would be constructed for the assembly in Figure 2 based on the system goals captured in Figure 4 and function allocation model described in Figure 5.

Furthermore, during the initialization process, the run-time model includes information pertaining to the component operational requirement for each component. If available, the explicit component operational requirement model is used. Otherwise, the implicit component operational requirement model is generated with nodes for the logical groups based on the expressions in Equation 4 and Table 3. Finally, each component operational requirement model is linked to the rest of the run-time model by parsing the component port interaction information captured in the assembly model.

Each node in the run-time model includes two boolean attributes that correspond to the *isActive* and *isUsable* properties mentioned in the previous sections. The *isActive* attribute captures the current state of the node: whether the node is currently active and if it is contributing to the system functions. In the case of a component, the *isActive* boolean attribute depends on its execution mode in the software component framework. For a logical group, the *isActive* property corresponds to whether any configuration based on the group contributes to any functionality. In the case of the function nodes, the *isActive* property captures

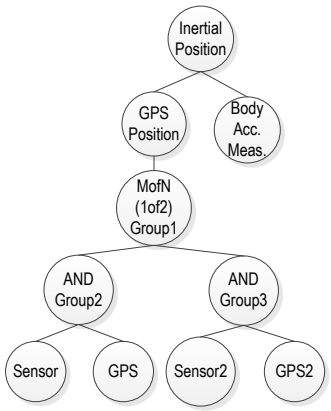

Fig. 10: Partial model in memory.

whether the functionality is currently being provided by the system. The $isUsable$ attribute reflects whether the node is able to provide service. In the case of a component, this corresponds to a component not having a fault. In case of a group node, the $isUsable$ property depends on the $isUsable$ property of its child nodes and whether they contain one or more component configurations that can provide the required service. Finally, in the case of a function node, the $isUsable$ property reflects whether the functionality can be provided by the system.

During the initialization of the run-time model, the value of the $isActive$ attribute of each component node is based on the initial execution mode of the component in the assembly model. The $isUsable$ property of each component node is initialized to true. The assumption is that during initialization, the components are not faulty. The $isActive$ and $isUsable$ properties of the nodes that correspond to the logical-groups (AND, ALT, MofN) are evaluated based on the relations described in Tables 1 and 2, using the value of the boolean attributes of the child nodes. For the purpose of evaluating these attributes in the function nodes, the functions are treated as AND-groups. These values are updated in a bottom-up fashion in the run-time model.

The next section deals with the deliberative reasoning process that handles the updates from the diagnoser and evaluates alternate configurations to restore the functionality.

5.2 The Deliberative Reasoner

The Deliberative Reasoner uses the run-time model to identify alternate configurations that can restore the affected functionality. Figure 11 provides an outline of the algorithm. It includes three steps: Init, Solve, and Update. The Figure shows the names of the procedures that are used in each step. A detailed discussion on the algorithm and the associated procedures can be found in [16].

Init: This step updates the run-time model based on the fault information. The procedures *MarkFaulty* and *VisitParent* ([16]) are used during this process. Procedure *MarkFaulty* sets the $isActive$ and $isUsable$ attribute of the faulty component node (in the run-time model) to false and invokes the Procedure *VisitParent*, which recursively traverses the graph bottom-up (child to parent) to evaluate and update the $isActive$ and $isUsable$ property of each node. The recursive update ends when the $isUsable$ property of a node continues to remain true. Such nodes are referred to as *reconfiguration nodes* and used in the next step.

Solve: This step attempts to identify an alternate configuration. It uses the Procedure *Reconfig* ([16]). The Procedure *Reconfig* starts from a *reconfiguration node* identified in the previous step (**Init**) and recursively traverses the graph top-down (parent to child) through the nodes that are still usable ($isUsable = true$) to identify a new set of component nodes that can be activated (by setting the $isActive$ property to true).

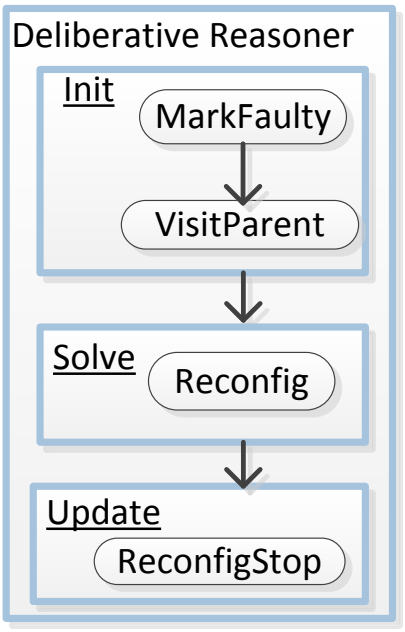

Fig. 11: Deliberative Reasoner.

Update: This step uses the Procedure *ReconfigStop* to update the run-time model to reflect the reconfigured system. The procedure *ReconfigStop* is invoked on each of the component nodes whose *isActive* property is true, starting first with the components identified in the previous step (**Solve**). A recursive traversal is performed to identify whether the node is contributing to any useful service; if it is not, it is deactivated ($isActive = false$). This is used to deactivate components that are not faulty, but do not contribute to any functionality in the new configuration.

Mitigation Commands: The altered state of components in the run-time model is used to generate mitigation or reconfiguration commands. These commands include:

1. `START`: Instructs a component to switch to the active mode. This command is issued when a component's *isActive* property has changed to true.
2. `STOP`: Instructs a component to switch to the inactive mode. This command is issued when a component's *isActive* property has changed to false.
3. `RESET`: Instructs a component to reinitialize itself. This command is issued when a component is faulty and an alternate configuration cannot be identified to restore functionality.
4. `REWIRE (ri,pc)`: Instructs a component to rewire its receptacle Interface (`ri`) and connect it to the appropriate provider interface in another component (`pc`). This command is issued when the component that hosted the original provider port is switched to inactive mode.

5.3 The Deliberative Reasoner and constraint solvers

The problem of identifying alternate configurations to restore the system functionality can be posed as a constraint satisfaction problem (CSP) that can be solved by external off-the-shelf constraint solvers. One way of dealing with large Boolean search spaces that has become very efficient in recent years is to use Boolean satisfiability (SAT) solvers [17]. Modern SAT solvers can efficiently solve problems containing thousands of variables and tens of thousands of clauses. Another approach is to use pseudo-Boolean satisfiability solvers, which are similar to SAT solvers but can handle cardinality constraints over the Boolean variables. Yet

another approach is to encode the problem as an integer linear arithmetic problem and use an SMT solver [31] to find a solution.

The run-time framework is designed in a way that allows the Deliberative Reasoner to be extended to use external, off-the-shelf solvers easily. Figure 9 shows the additional steps that can be added to the Deliberative Reasoner so that it can use such off-the-shelf solvers. These steps include:

Problem Encoding: This step involves identifying the variables and their associated constraints, and translating these into the domain of the external solver. In our case, the variables and constraints include those identified in the run-time model and their underlying relationships captured in the Equations and Tables in section 4. Further, the encoded problem accounts for the faulty components as well as the current component-states. This is to ensure that the resulting solution does not include the faulty components and is close to the original component configuration.

Execution: This step involves executing the off-the-shelf solver. Due to the nature of the environment where the ACM framework is deployed, off-the-shelf solvers that provide a C/C++ library that allows programmatic access for execution in a Linux-based operating system are preferred. The solver is expected to provide an API that allows the deliberative reasoner to input the satisfaction problem, execute the solver and then obtain the solution.

Solution Decoding: In this step, the resulting solution from the solver is translated to the original problem domain. In our case, this corresponds to setting the $isActive$ and $isUsable$ properties of the nodes in the run-time model based on the solution from the solver.

Once these steps are completed, the Deliberative Reasoner can invoke the **Update** step (described earlier) to update the run-time mode with the new result, verify, and if required, post-process the result. The following section describes our approach to integrating one specific type of solver, a boolean Satisfiability (SAT) solver, in the run-time framework.

6 Deliberative Reasoning with a SAT solver

This section describes the process of integrating the Deliberative Reasoner with a SAT solver. This involves generating a Boolean satisfiability encoding of the original search problem, invoking the solver and decoding the solution (if found) produced by the solver. For practical reasons, modern SAT solvers work with a set of clauses in conjunctive normal form (CNF). CNF consists of the conjunction (i.e., logical AND) of a set of clauses, where each clause is a disjunction (i.e., logical OR) of literals (Boolean variables). Therefore, using a SAT solver involves (1) encoding the information captured in the runtime model into a set of clauses in CNF, (2) invoking a SAT solver to try to find a satisfying solution, and (3) decoding the solution (if found) from the SAT solver so that the runtime model can be updated and corresponding mitigation commands can be issued.

6.1 Identifying variables and their relationships

The runtime model, which is the basis of the Deliberative Reasoner, includes boolean properties ($isActive$ and $isUsable$) for the functions, the group nodes in the function allocation model and the components. In addition, it also includes the information about the component's operational requirement model: the $isActive$ properties of the consumer and requires ports of each component and the group nodes associated with this component operational requirement model. Because these variables have boolean values, they naturally align with a Boolean variable in a satisfiability problem. Table 4 lists these categories of Boolean variables and the associated Boolean expressions used to evaluate them.

6.2 Identifying constraints

The primary set of constraints are expressed over the Boolean variables assigned to the nodes in the function allocation model: the functions, the logical groups (AND, ALT, MofN) and the components. Each logical

Table 4: Boolean variables for the SAT problem

Type	Variable category	Is Free Variable	Related Expression
Component, comp	V_{comp}	Yes	Set by solver*
Component, comp	$V_{COR_{comp}}$	No	COR(comp)
Consumer,c	V_c	No	*isActive* (c)
Requires,r	V_r	No	*isActive* (r)
ALT-Group,x	V_x	No	*isActive* (x)
MofN-Group,m	V_m	No	*isActive* (m)
AND-Group,a	V_a	No	*isActive* (a)
Function,f	V_f	No	*isActive* (f)

* - corresponds to the *isActive* (comp)

group is encoded into one or more Boolean constraints such that the encoding preserves the semantics of the logical grouping (Table 2). In the case of function nodes, the constraints are generated by assuming the function to be a logical AND grouping.

The next set of constraints are related to the component operational requirement model and the interactions captured in the assembly model. One or more Boolean constraints are generated to capture the implication relationship between the *isActive* property of a component and its operational requirement expression (Equation 3). If a component needs to be active (i.e., *isActive* is set to true), then its corresponding component operational requirement expression should evaluate to true. This also involves generating the clauses corresponding to the logical groups found in the component operational requirements model (Tables 3).

$$V_{comp} \implies V_{COR_{comp}} \tag{15}$$

Additional Boolean constraints are imposed based on the functional requirement trees. These translate to a constraint stating that the logical AND of the *isActive* property of all the root functions should be true (Equation 16). Further, in a run-time component assembly, when the diagnoser reports that certain components are faulty, this information needs to be encoded as additional constraints. The Boolean variable associated with the *isActive* property of each faulty component is set to false.

$$\bigwedge_{i \in RF} V_{f_i} \tag{16}$$

where

RF is the set of root functions

V_{f_i} is Boolean variable associated with i^{th} function f_i

6.3 Problem encoding: transforming to CNF

The Boolean expressions capturing the relationships between the Boolean variables and the constraints need to be converted into conjunctive normal form (CNF) before they are given to a SAT solver. Such a translation of Boolean expressions into CNF is discussed in [47]. Our problem includes three additional relations, MofN, ALT and Implies, which are translated into CNF using the same encoding techniques described in [47]; we provide an overview below.

Handling ALT: While some SAT solvers have special constructs to express ALT directly, this is not supported by most solvers. Hence, the ALT expressions are translated into CNF in the following way. Intermediate Boolean variables are introduced, each of which represents a distinct valid combination of variables that can satisfy the ALT. Equation 18 lists the n intermediate Boolean variables introduced and their distinct valid combination (conjunction) for a successful outcome of the ALT expression 17 with n components ($comp_1$, $comp_2$,..., $comp_n$). Note that each of the expressions captured by the intermediate

variables allows exactly one child node to be true, while all other child nodes are expected to be false. The disjunction of these intermediate Boolean variables (Equation 19) is an equivalent Boolean expression that replaces the original ALT expression (Equation 17). Since this expression (Equation 19) involves only conjunctions and disjunctions, it can be easily transformed into the CNF format using the strategies discussed in [47].

$$V_x = ALT(comp_1, comp_2, ..., comp_n) \tag{17}$$

$$\begin{gathered} V_{x_1} = comp_1 \wedge \neg comp_2 \wedge ... \wedge \neg comp_n \\ V_{x_2} = \neg comp_1 \wedge comp_2 \wedge ... \wedge \neg comp_n \\ ... \\ ... \\ V_{x_n} = \neg comp_1 \wedge \neg comp_2 ... \wedge comp_n \end{gathered} \tag{18}$$

$$V_x = V_{x_1} \vee V_{x_2} \vee ... \vee V_{x_3} \tag{19}$$

Handling MofN: For MofN, *at least* M out of the N child nodes should be true. The approach adopted to capture this relationship in the form of a Boolean expression with just AND and OR operations is similar to the case of ALT. Each distinct valid combination that satisfies the MofN criteria is represented explicitly by an intermediate Boolean variable. However, since MofN does not require that *exactly* M out of N variables should be true, but requires that *at least* M out of N be true, the expressions of the intermediate variables do not deal with the variables that are false. The creation of the intermediate variables (and their expressions) involves identifying each of the $C(M, N)$ combinations where M variables are true. Each of these distinct combinations is expressed as a conjunction of the true variables, and the result is assigned to an intermediate variable. The final step is to rewrite the MofN expression as a disjunction of all the intermediate variables. Equation 20 captures this process for a simple example involving three variables (N=3) and MofN with M=2. Equation 21 represents the more generic case where $N = n$ and $M = m$.

$$\begin{gathered} V_m = MofN(comp_1, comp_2, comp_3) \\ V_{m_1} = comp_1 \wedge comp_2 \\ V_{m_2} = comp_1 \wedge comp_3 \\ V_{m_3} = comp_2 \wedge comp_3 \\ V_m = V_{m_1} \vee V_{m_2} \vee V_{m_3} \end{gathered} \tag{20}$$

$$\begin{gathered} V_m = MofN(comp_1, comp_2, ..., comp_n) \\ V_{m_1} = comp_1 \wedge comp_2 \wedge ... \wedge comp_{m-1} \wedge comp_m \\ V_{m_2} = comp_1 \wedge comp_2 \wedge ... \wedge comp_{m-1} \wedge comp_{m+1} \\ V_{m_3} = comp_1 \wedge comp_2 \wedge ... \wedge comp_{m-1} \wedge comp_{m+2} \\ ... \\ V_{m_{n-m}} = comp_1 \wedge comp_2 \wedge ... \wedge comp_{m-1} \wedge comp_n \\ ... \\ V_{C(M,N)} = ... \\ V_a = V_{a_1} \vee V_{a_2} \vee ... \vee V_{a_{C(M,N)}} \end{gathered} \tag{21}$$

Handling implication: In the case of the implies operator, an implication constraint exists between the component's *isActive* property and the component's operational requirement condition (Equation 15). This implication relationship between the Boolean variables V_{comp} and $V_{COR_{comp}}$ needs to be expressed in CNF. The CNF translation of this implication constraint is expressed as a Boolean constraint (Equation 22) that needs to be satisfied by the SAT solver.

$$V_{COR_{comp}} \vee \neg V_{comp} \tag{22}$$

Once these transformations are applied to the ALT, MofN and Implies relations, the resulting CNF formula can be given to the SAT solver.

6.4 Execution and decoding the SAT solver results

Once the steps detailed in the previous sections are performed, the SAT solver can be given the following information: (1) the variables (literals) and (2) the relationships and constraints (CNF clauses) capturing the original problem. Two additional pieces of information also need to be encoded and given to the solver before it is started: (1) the faulty components, and (2) the current state of the components.

Handling faulty components: It is important that the SAT solver is made aware of faulty components. Otherwise the solution output from the SAT solver could continue to include using the services of the faulty component(s) to achieve the result (i.e., provide services). Additional constraints (clauses) are added to the original problem expressed in CNF to capture the information pertaining to the faulty components. For each faulty component, *comp*, a clause that captures the constraint expressed in Equation 23 is added. This tells the SAT solver that the state of this component should be false in the final solution.

$$\neg comp \tag{23}$$

where *comp* is a faulty component

Handling component states: An additional property that is desired from the SAT solution is that the process of identifying a new solution (for reconfiguration) should take into consideration the current component configuration in the assembly, with the goal that if possible, healthy (non-faulty) and active components that are associated with services that are unaffected should not be reconfigured. The solution should involve minimal reconfiguration if such a path exists. In order to do this, the SAT solver used in this exercise (Cryptominisat v.2.9.1 [5]), allows one to specify a set of assumptions for the state of one or more literals (which correspond to the state of the appropriate component). Since there is a one-to-one association between the literals in the defined SAT problem and the *isActive* property of the components (V_{comp}), this step involves identifying the literal that corresponds to each healthy active component and adding an assumption to the SAT problem that this literal must be true. The components that are currently stopped or have been detected as faulty are not considered in this process.

Executing the solver and decoding the solution: Once all of the steps above have been performed, the SAT solver can be executed to verify an initial system configuration (the case where there is no-fault) or to identify an alternate configuration in the case of a fault. The verification of an initial solution is performed by invoking the SAT solver and setting the assumptions on the Boolean literal corresponding to each active component. If the SAT solver reports that the solution is satisfactory (SAT), then the initial configuration is valid. Otherwise, the SAT solver results need to be analyzed to identify problems either in the initial configuration or in the models.

Whenever a fault report is obtained from the diagnosis engine, an additional clause based on Equation 23 is added for each faulty component. Also, the states of the faulty components are not encoded as assumptions while invoking the SAT solver. The SAT solver is invoked with assumptions only for the states of healthy and active components. The SAT solver results are then analyzed for a reconfiguration solution.

Handling UNSAT solutions: In the case where the SAT solver reports an unsatisfiable problem (UNSAT), then the assumptions given to the SAT solver are relaxed. The SAT solver is queried for the conflicts, and the assumptions pertaining to the literals reported in the conflict are removed. The assumptions for the rest of the healthy and active components are set and the SAT solver is invoked again. In some cases, all of the assumptions may need to be removed. This is because it is possible that for obtaining a valid configuration, all of the currently active components need to be deactivated and a completely new set of components needs to be activated. However, it is also possible that after removing all assumptions, the solver still reports an UNSAT solution. In such cases, this implies that there is no redundant configuration that can restore the functionality, and the only possible mitigation action is to reset the faulty component(s).

[5] http://www.msoos.org/cryptominisat2/

Table 5: Boolean variables for the SAT problem in Example 1

Variable(s)	Category*	Description
V_{GPS},V_{GPS2}, V_{Sensor}, $V_{Sensor2}$, $V_{NavDisplay}$	V_{comp}	Variables corresponding to a component's *isActive* property. Set by solver.
$V_{COR_{GPS}}$, $V_{COR_{GPS2}}$, $V_{COR_{NavDisplay}}$	$V_{COR_{comp}}$	Variables corresponding to a component's Operational Requirement.
V_{dat}	V_c	Variable corresponding to the *isActive* property of Consumer (*dat*) in NavDisplay.
V_{gps}	V_r	Variable corresponding to the *isActive* property of Requires (*gps*) in NavDisplay.
V_{pos}	V_f	Variable corresponding to the *isActive* property of GPS Position Function in Function Allocation Model.

* - Category is described in Table 4.

Handling SAT solutions: When the SAT solver returns with an output indicating satisfiability (SAT), the results from the SAT solver can be used to reconfigure the system. The one-to-one correspondence between the literals in the SAT solver problem and the Boolean variables associated with the component states (the *isActive* property) allows for direct application of the SAT solver solution. The SAT solver results are used to update the component states (*isActive* property) in the run-time model as if the results were obtained from the Reconfig procedure of the Deliberative Reasoner. The update algorithm is run on the entire model so that the *isActive* property of each node is updated. At this point, the ReconfigStop procedure is also invoked to identify any component that is active but not contributing to any functionality. Such components are marked to be deactivated. This can happen in SAT solver solutions because a valid solution may include a component which is active, but whose parent groups are not active because the required set of child components is not active. Finally, all the components where the *isActive* properties have changed are identified and appropriate mitigation commands issued. Additionally, mitigation commands for rewiring required interfaces are also sent.

6.5 Smaller examples

Example 1 This section illustrates the approach of using a SAT solver within the Deliberative Reasoner. The example deals with the assembly model in Figure 2 configured with the goals described in the system goals model in Figure 4 and the function-allocation model described in Figure 8. It also uses the implicit component operational requirement for all the components in the assembly. Note that the expressions for the Implicit Component Operational Requirements for GPS, GPS2 and NavDisplay were derived earlier in Equations 5 and 13.

The Boolean variables associated with the SAT problem are listed in Table 5. The relationships and the constraints expressed by these variables are enumerated in Equations 24 and 25. For equations using the ALT and Implies relations, the transformations detailed in section 6.3 are applied to produce the Equations 26, 27.

$$\begin{gathered}\text{Satisfy} : V_{pos} \\ \text{Subject to:} \\ V_{pos} = AND(V_{NavDisplay}) = V_{NavDisplay}\end{gathered} \tag{24}$$

$$
\begin{aligned}
V_{NavDisplay} &\implies V_{COR_{NavDisplay}} \\
V_{COR_{NavDisplay}} &= ALT(V_{GPS}, V_{GPS2}) \\
V_{GPS} &\implies V_{COR_{GPS}} \\
V_{GPS2} &\implies V_{COR_{GPS2}} \\
V_{COR_{GPS}} &= V_{Sensor} \\
V_{COR_{GPS2}} &= V_{Sensor2}
\end{aligned} \tag{25}
$$

$$
\begin{aligned}
V_{COR_{NavDisplay}} &= ALT(V_{GPS}, V_{GPS2}) \\
\text{Let } V_{temp_1} &= V_{GPS} \wedge \neg V_{GPS2} \\
\text{Let } V_{temp_2} &= \neg V_{GPS} \wedge V_{GPS2} \\
V_{COR_{NavDisplay}} &= V_{temp_1} \vee V_{temp_2}
\end{aligned} \tag{26}
$$

$$
\begin{aligned}
V_{COR_{NavDisplay}} \vee \neg V_{NavDisplay} \\
V_{COR_{GPS}} \vee \neg V_{GPS} \\
V_{COR_{GPS2}} \vee \neg V_{GPS2}
\end{aligned} \tag{27}
$$

Equation 28 captures the final set of formulas in CNF that are fed to the SAT solver. Each line in this formula captures a relation that needs to be true.

$$
\begin{aligned}
&\text{Satisfy: } V_{pos} \\
&\text{Subject to :} \\
&\neg V_f \vee V_{NavDisplay} \ \wedge \\
&V_{COR_{NavDisplay}} \vee \neg V_{NavDisplay} \ \wedge \\
&V_{COR_{NavDisplay}} \vee \neg V_{temp_1} \ \wedge \\
&V_{COR_{NavDisplay}} \vee \neg V_{temp_2} \ \wedge \\
&\neg V_{COR_{NavDisplay}} \vee (V_{temp_1} \vee V_{temp_2}) \ \wedge \\
&\neg V_{temp_1} \vee V_{GPS} \ \wedge \\
&\neg V_{temp_2} \vee V_{GPS2} \ \wedge \\
&V_{COR_{GPS}} \vee \neg V_{GPS} \ \wedge \\
&V_{COR_{GPS2}} \vee \neg V_{GPS2} \ \wedge \\
&\neg V_{COR_{GPS}} \vee V_{Sensor} \ \wedge \\
&\neg V_{COR_{GPS2}} \vee V_{Sensor2}
\end{aligned} \tag{28}
$$

The set of equations 28 were input to a SAT solver and the assumptions related to the initial system state were set. This included setting the literals associated with all component nodes except GPS2 to true. This was because the GPS2 component was set to semi-active execution mode, while the rest of the components were set to active-mode.

As a first step, the external engine (SAT solver) was invoked to verify that the system state based on the initial component configurations can provide the required functionality. Once this was confirmed, the next step involved injecting a fault into the Sensor component. This was correctly diagnosed by the Diagnosis Engine component, and resulted in the Deliberative Reasoner being triggered. An additional constraint ($\neg V_{Sensor}$), reflecting the non-availability of the faulty sensor component was added to the set of clauses in Equation 28. The current state of the healthy components (GPS, NavDisplay and Sensor2) was fed as assumptions to the SAT solver. The SAT solver produced a SAT solution with a configuration that involved turning off the Sensor and GPS components and activating the GPS2 component, while retaining the active state of the Sensor2 and NavDisplay components

The resulting reconfiguration comands from the Deliberative Reasoner included

- STOP Sensor, STOP GPS
- START GPS2

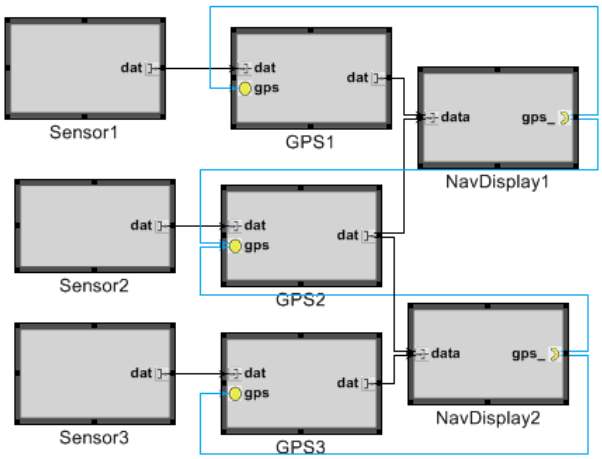

Fig. 12: Example 2: Assembly Model

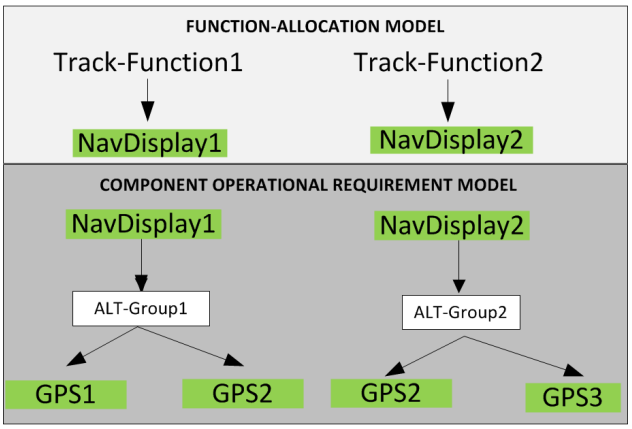

Fig. 13: Example 2: Equivalent Core Function Allocation Model and the Implicit Component Operational Requirements

– REWIRE NavDisplay: New Provider GPS2

The resulting configuration was able to restore the functionality. This exercise demonstrated a simple example of using the Deliberative Reasoner with a SAT solver to reproduce the results obtained using the native Deliberative Reasoner discussed in [16]. Further, this example used a more concise function allocation model and derived the component operational requirement relations from the assembly model.

Example 2 In this example, the Deliberative Reasoner with a SAT solver is tested on a more complicated model. The assembly model is described in Figure 12. The function allocation model and component operational model are shown in Figure 13. The system goal requires the functions Track-Function1 and Track-Function2 to be supported. The complication in this model is brought about by the ALT-Groups. In order for the ALT-group to be active, exactly one of its child nodes must be active (Table 2). More importantly, this example (Figure 13) includes a component, GPS2, that is featured in two ALT-Groups (ALT-Group1 and ALT-Group2), each of which has to be active to satisfy the system goals (Track-Function1 and Track-Function2).

The initial system state is as follows

– *Active Components:* Sensor1, Sensor2, Sensor3, GPS1, GPS3, NavDislay1 and NavDisplay2.
– *Active Functions:* TrackFunction1, TrackFunction2, and Tracking.

– *Stopped Components:* GPS2.
– *RMI Wiring:* NavDisplay2 to GPS3, and NavDisplay1 to GPS1 .

The Deliberative Reasoner with SAT was able to verify the initial state of the system. A fault was introduced in the Sensor3 component. The Deliberative Reasoner using the SAT solver issued the following reconfiguration commands

– STOP Sensor3, STOP GPS3, STOP Sensor1, STOP GPS1.
– START GPS2.
– REWIRE NavDisplay2 to GPS2.
– REWIRE NavDisplay1 to GPS2.

After reconfiguration, the system state was as follows

– *Active Components:* Sensor2, GPS2, NavDislay1, and NavDisplay2.
– *Faulty Components:* Sensor3.
– *Stopped Components:* Sensor1, Sensor3, GPS1, and GPS3
– *Active Functions:* TrackFunction1, TrackFunction2, and Tracking.
– *RMI Wiring:* NavDisplay2 to GPS2, and NavDisplay1 to GPS2 .

It can be seen that the reconfiguration solution obeys the ALT constraint. It switches off the faulty component Sensor3 and GPS3 that relies on Sensor3. Further, to satisfy the ALT constraint it switches off GPS1 and activates GPS2. Furthermore, it switches off the Sensor1 component as its services are no longer required.

6.6 A Larger Case Study: IMU

We use a larger example of an inertial measurement unit (IMU) to present further details about the Deliberative Reasoner. An IMU is a software assembly that uses accelerometers and GPS units to track the inertial position in an avionics system. The IMU assembly (see Figure 14 includes redundant configurations with four kinds of subsystems, which are described below.

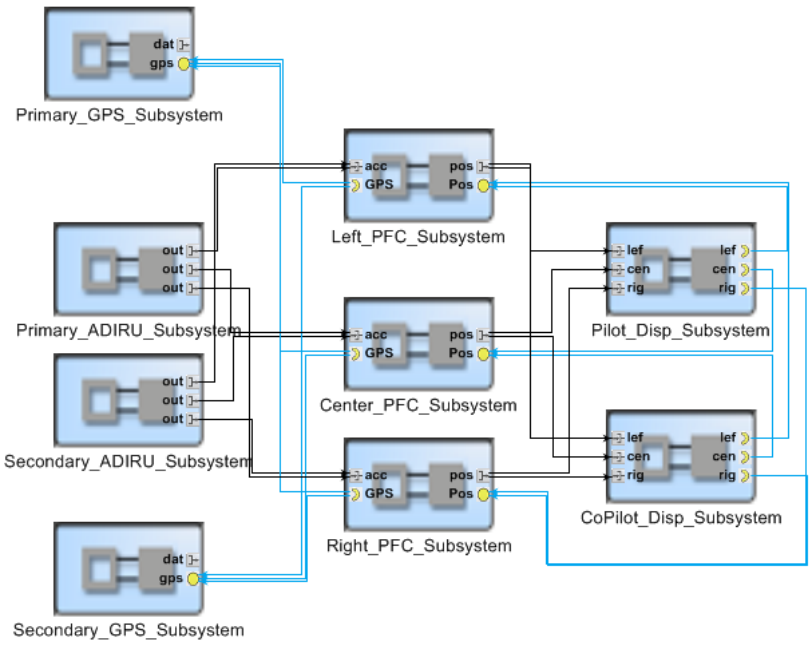

Fig. 14: The layout of the inertial measurement unit

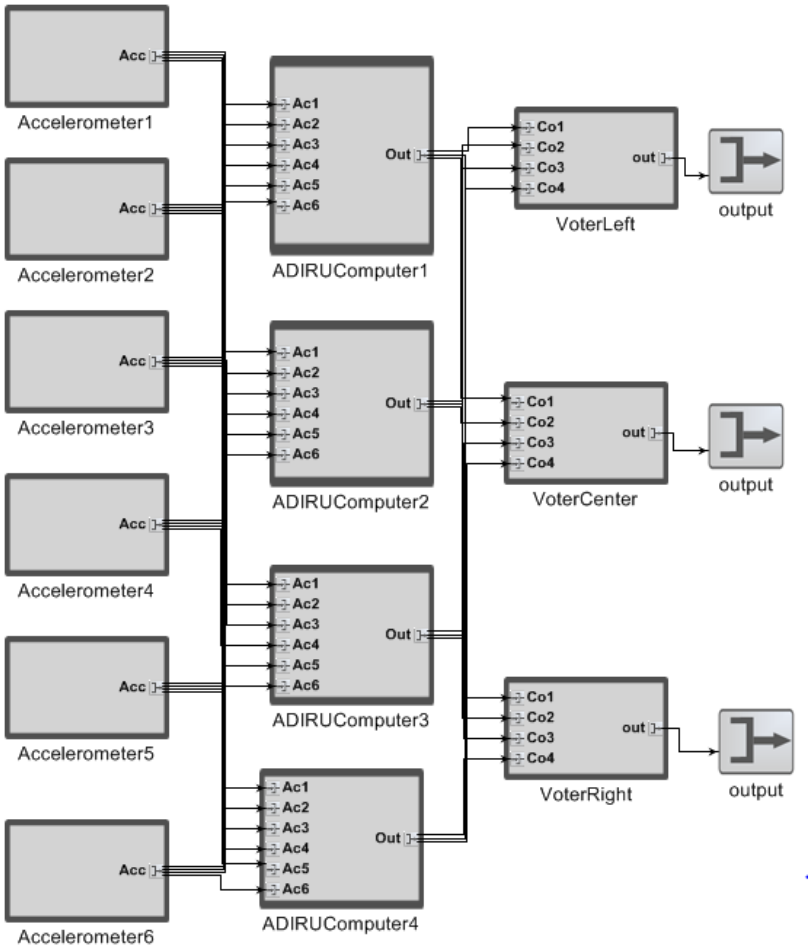

Fig. 15: ADIRU.

Air Data Inertial Reference Unit (ADIRU) The IMU system includes a primary and a backup ADIRU subsystem. The architecture of the ADIRU subsystem (see Figure 15) is based on the ADIRU used on a Boeing 777 aircraft [29,43]. It includes six accelerometer components, four `ADIRU Processor` components and three `Voter` components. Each accelerometer component publishes data to all the four ADIRU processors by reading an emulated sensor. Each ADIRU processor uses a set of linear regression equations to periodically estimate the body acceleration and publish it to the voter components. Each voter component votes upon the body-axis estimate of the ADIRU processors and publishes the result to the `Primary Flight Computer` (PFC) components.

GPS Subsystem The IMU system includes a primary and backup GPS subsystem (Figure 16) which includes two components. The GPS receiver component emulates a software sensor providing the hardware readout to the GPS processor component that implements a Kalman Filter. On each update, the GPS processor notifies the PFC components.

Primary Flight Computer (PFC) Subsystem The PFC subsystem shown in Figure 17 emulates the flight computer which uses the body acceleration data fed by the ADIRU to track the airplane's inertial position. The IMU system is configured with three PFC subsystems: left, right, and center - that actively receive the input from a Voter component in the ADIRU subsystem (see Figure 14). Given that the inertial system using the body acceleration values tends to drift over time, the PFC NavFilter component uses a receptacle port to fetch the more accurate but slowly refreshing GPS data.

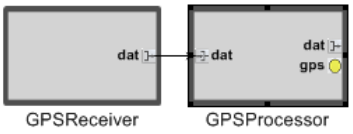

Fig. 16: GPS Subsystem

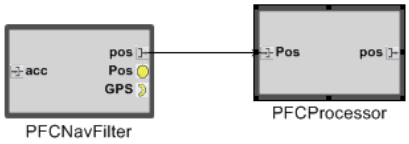

Fig. 17: PFC Subsystem.

Display subsystem The Pilot and Co-Pilot Display subsystems (Figure 18) receive update notifications from the three PFC subsystems. The Display component periodically fetches the updated data (through its requires ports) from each of the PFC components and displays a median value.

System goals for IMU The IMU system goal (see Figure (19) is to provide inertial tracking functionality. Inertial tracking depends on determining gps position as well as position tracking. Position tracking, in turn, depends on the ability of the system to determine the body acceleration.

Function Allocation in IMU The mathematical relations that capture the function allocation models in the IMU system are listed below.

- The `GPS-Position` functionality requires exactly one of the GPS subsystems, i.e. $GPSPosition \rightarrow$ `EXACTLY`$(1, Primary, Secondary\ GPSsubsystem)$.
- The `Position-tracking` functionality requires at least 1 of the PFC subsystems, i.e. $PositionTracking \rightarrow$ `ATLEAST`$(1)(Left, Center, RightPFCSubsystem)$.
- The `Body Acceleration` functionality requires exactly 1 of the ADIRU subsystems, i.e. $BodyAcceleration \rightarrow$ `EXACTLY`$(1, Primary, SecondaryADIRUsubsystem)$.

Component operational requirements in IMU The following are the explicitly specified component operational requirements in the IMU system.

- The `Display` component needs at least one of its consumers to be active: $\rightarrow$ `ATLEAST`$(1)(left, right, center)$ consumer.
- The `ADIRU Processor` component requires at least 4 of its 6 consumers to be active : $\rightarrow$ `ATLEAST`$(4)(Allconsumers)$.
- The `Voter` component in the ADIRU subsystem requires at least 2 of its 4 consumers to be active : $\rightarrow$ `ATLEAST`$(4)(Allconsumers)$.

The operational requirement for the other components is derived implicitly using Equation 4.

Redundant configurations in IMU The IMU subsystem includes 9 subsystems (2 ADIRU subsystems, 2 GPS subsystems, 3 PFC subsystems and, 2 Display subsystems) with a total of 40 components between them. Based on the function-allocation and component operational requirement models, the number of alternate configurations can be identified as follows:

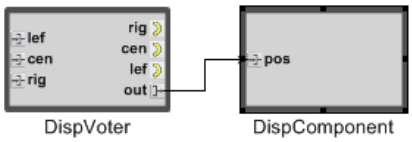

Fig. 18: Display Subsystem

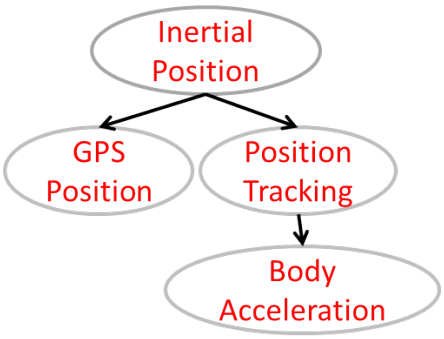

Fig. 19: IMU Functional Requirements.

- 2 alternate configurations support GPS-Position functionality (using Primary or Secondary GPS subsystem).
- 7 alternate configurations support Position-Tracking functionality (At least 1 of the 3 PFC subsystems needs to be active i.e. ${}^3C_3 +^3 C_2 +^3 C_1) = 7$).
- 2 alternate configurations support Body-Acceleration functionality (Primary or Secondary ADIRU).
- 37 alternate configurations of the accelerometer components in each ADIRU subsytem (4 out of the 6 accelerometers required i.e.${}^6C_6 +^6 C_5 +^6 C_4) = 37$).
- 17 alternate configurations of the ADIRU processor components in each ADIRU subsystem(2 out of the 4 processors required i.e.${}^4C_4 +^4 C_3 +^4 C_2 = 17$).

The total possible configurations is a product of the number of alternate configurations available in each case above, i.e. 17612 $(= 2 * 7 * 2 * 37 * 17)$.

IMU health management using Deliberative Reasoner with a SAT solver The whole IMU assembly was deployed on four hosts using one core clocked at 2.4 GHZ in each hosts. The SLHM components associated with the assembly were deployed on a separate dedicated core. A Deliberative Reasoner with a SAT solver [6] was used to identify alternative configurations to mitigate fault-effects and restore the IMU functionality. The CNF encoding for the IMU system included 493 variables and 1776 clauses. The system was initialized with the components in the following subsystems set to active mode: Primary ADIRU, Primary GPS, All PFCs and all Display subsystems. The secondary GPS and ADIRU subsystems were set to inactive mode. The initial configuration was verified using the SAT solver in 4.228 milliseconds. Thereafter, the above setup was tested for a specific fault scenario. The sequence of faults, the time for computing reconfiguration commands for each fault and the reconfiguration commands issued after each fault are listed in Table 6.

When the first fault is triggered (Accelerometer6 in Primary ADIRU subsystem), the reconfiguration engine correctly identified a solution close to the original configuration, i.e., use the original configuration but turn off the faulty component (Accelerometer6). It did the same when another accelerometer turned faulty (Accelerometer 5), as the ADIRU can tolerate up to two Accelerometer faults. When the third Accelerometer failed (Accelerometer4), the reconfiguration engine correctly identified that the Primary ADIRU was no longer capable of supporting the desired functionality and switched to using the secondary ADIRU subsystem.

[6] http://www.msoos.org/cryptominisat2/, v2.9.1

Table 6: Results of using the deliberative reasoner with the SAT solver on the IMU assembly

Sequence No.	Fault	Compute Time (ms)	Reconfig Commands
1	Primary ADIRU Accelerometer6	2.98	STOP (Primary ADIRU Accelerometer6)
2	Primary ADIRU Accelerometer5	3.15	STOP (Primary ADIRU Accelerometer5)
3	Primary ADIRU Accelerometer4	2.082	STOP (Primary ADIRU Accelerometer4) STOP (Primary ADIRU subsystem) START (Secondary ADIRU subsystem)
4	Primary GPS Processor	4.720	STOP (Primary GPS Receiver) STOP (Primary GPS Processor) START (Secondary GPS Receiver) START (Secondary GPS Processor) REWIRE (Left PFC, GPS Data Source, Secondary GPS Processor) REWIRE (Right PFC, GPS Data Source, Secondary GPS Processor) REWIRE (Center PFC, GPS Data Source, Secondary GPS Processor)

In the case of the fault in the GPS component, the reconfiguration engine switched to the Secondary GPS subsystem and rewired the PFCs to use the provider ports in the Secondary GPS subsystem.

7 Deliberative reasoning using a Pseudo-Boolean solver

This section describes the process of integrating the Deliberative Reasoner with a pseudo-Boolean (PB) solver. As in the case of integrating with a SAT solver (section 6), this involves (1) translating the information captured in the runtime model into a set of pseudo-Boolean clauses, (2) invoking a PB solver to try to find a satisfying solution, and (3) decoding the solution (if found) from the PB solver back into the runtime model and issuing the appropriate mitigation commands.

7.1 Encoding for using PB solvers

The PB solvers are extensions to SAT solvers. In addition to handling the regular Boolean satisfiability constraints, they can handle cardinality constraints. The variables and constraints that need to be input to the PB solver are exactly the same as the ones described in the case of SAT solver (see Sections 6.1 and 6.2). The encodings for the AND-groups, functions, component operational requirements, faulty components and component states are quite similar and can be derived from the CNF encodings presented in Section 6.3. However, the pseudo-Boolean encoding for the ALT and MofN groups is much simpler. The following paragraphs describe the PB encoding for each of these cases.

Handling AND: The PB encoding for handling the AND relation between two variables V_1 and V_2 is captured in Equation 29, where V_a represents the AND relation itself. Equation 30 captures the corresponding CNF encoding. The one-to-one relationship between the two translations is evident. The PB encoding for an AND relation involving N variables - $V_1, V_2, ..., V_N$ is captured in Equation 31.

$$\begin{aligned} -V_a + V_1 &\geq 0 \\ -V_a + V_2 &\geq 0 \\ V_a - V_1 - V_2 &\geq -1 \end{aligned} \tag{29}$$

$$\begin{aligned} \neg V_a \vee V_1 \ \wedge \\ \neg V_a \vee V_2 \ \wedge \\ V_a \vee \neg V_1 \neg V_2 \ \wedge \end{aligned} \tag{30}$$

$$\begin{aligned} -V_a + V_1 \geq 0 \\ -V_a + V_2 \geq 0 \\ ... \\ -V_a + V_N \geq 0 \\ V_a - V_1 - V_2 - ... - V_N \geq -N + 1 \end{aligned} \tag{31}$$

Handling ALT: The encoding of the ALT-groups is greatly simplified in the case of PB solvers. An ALT relationship between N variables - $V_1, V_2, ..., V_N$ is captured by Equation 32 where V_a represents the result of the ALT relation.

$$V_a - V_1 - V_2 - ... - V_N = 0 \tag{32}$$

Handling MofN: Like ALT-groups, the PB encoding for MofN groups is much simpler than the corresponding CNF encoding. An MofN relationship between N variables - $V_1, V_2, ..., V_N$ is captured by Equation 33 where V_a represents the result of the MofN relation.

$$\begin{aligned} -M \cdot V_a + V_1 + V_2 + ... + V_N \geq 0 \\ (N - M + 1) \cdot V_a - V_1 - V_2 - ... - V_N \geq -M + 1 \end{aligned} \tag{33}$$

Maximal activation in MofN: While the MofN relation described above captures the requirement that a minimum of M nodes are required to activate an MofN relation, we are interested in a maximal solution, i.e., when an MofN node is activate, we want to activate as many of its child nodes as possible. This is enforced through a minimization constraint supported by PB solvers. Equation 34 captures the minimization constraint for each MofN, where $V_a, V_1, V_2, ..., V_N$ represent the state of the variables in Equation 33.

$$minimize \ \{V_a \cdot (N \cdot V_a - V_1 - V_2 - ... - V_N)\} \tag{34}$$

Handling implies: The PB encoding for the implies relationship between the *isActive* property of a component and its component operational requirement is similar to its CNF encoding 22. It is shown in Equation 35, where V_{comp} and $V_{COR_{comp}}$ represent the *isActive* property and component operational requirement for a component *comp*.

$$V_{comp} - V_{COR_{comp}} = 0 \tag{35}$$

Handling faulty components: An additional clause (Equation 36) is added for each faulty component *comp*.

$$V_{comp} = 0 \tag{36}$$

Handling component states: As in the case of SAT solvers, the component states are provided to the PB solver as assumptions. The literals corresponding to the healthy active components are set to true.

Handling functions and solutions: As in the case of the CNF encoding, the AND relationship described in Equation 31 is used to capture the inter-dependencies between the functions (in the system goal model)

as well as the relationship between the root functions and the solution variable (V_s). The constraint on the solution variable, V_s is expressed by the Equation 37.

$$V_s = 1 \tag{37}$$

The relations captured in this section were used to set up the pseudo-Boolean problem for illustrative examples similar to those in Section 6.5. A pseudo-Boolean solver, MINISAT^{+}[7], was used to solve the problem. The resulting solutions appeared to capture the correct reconfiguration strategies. A more detailed discussion of these results is planned as future work.

8 Discussion and Future work

The search process of the native Deliberative Reasoner (summarized in section 5.2, discussed in detail in [16]) is rather simple and straightforward. The search process is extremely effective, in that it deals only with the affected portion of the graph and performs a local-search as close as possible to the affected nodes in the original configuration. In certain scenarios, such as the exclusivity relationship imposed by ALT-groups, the SAT solver is useful, as demonstrated by the results of the examples in the previous sections. Setting an assumption on the literals associated with the active-healthy components allows us to direct the SAT solver to find an existing solution that does not affect the functioning parts of the system. The encoding for the pseudo-Boolean solvers (presented in Section 7) is much more direct and simple than the CNF encoding for the SAT solvers. This is especially true for the MofN relation. Our tests, thus far on simple illustrative examples, have been very promising, and we plan to test this approach on larger systems. We are also exploring the use of other solvers, such as SMT solvers [31].

Our current approach lacks the ability to use a solver selectively for a subset of the problem. For example, it is possible to extend the approach and formulate the problem so that it deals exclusively with the functions and groups that are related to the affected components (those affected due to discovered faults). This can improve scalability in very large systems.

Additionally, the current approach uses either the *Solve* step of the Deliberative Reasoner or uses an external solver for the entire problem. If the problem can be broken down through offline-analysis or analysis at initialization time, it should be possible to use an array of solvers for the different parts of the problem. This could involve the quick and simple native Deliberative Reasoner as well. The hybrid approach could also involve abstracting the problem sent to the external solver, so that the simpler native Deliberative Reasoning approach is used to post-process additional results based on the configuration supplied by the external solver.

Apart from solutions that enhance runtime performance, additional tools to help at design-time should be created as well. These tools can help verify the consistency of the system on an incremental basis so that problems with the design-time specifications can be fixed.

9 Related research

The work described in this paper generally falls into two categories (a) runtime monitoring, and (b) self-adaptive software systems. The difference clearly lies in monitoring and detection vs. monitoring and detection and mitigation. The work presented in this paper is focusing primarily on the mitigation aspects. However, it uses our work in the area of monitoring, detection, and fault isolation.

9.1 Run-time Monitoring and Detection

Methods for run-time detection of faults can be classified either as acceptance-based testing, or comparison-based testing. The former involves monitoring a component or subsystem with respect to some acceptance criteria, while the latter uses multiple executions whose results are then compared.

Pike et. al. described the Copilot run-time monitor for periodic tasks in embedded systems in [35]. They described their approach for establishing a run-time monitoring framework where monitors can be scheduled

[7] http://minisat.se/MiniSat+.html

by integrating the monitor executions in the system schedule such that all the task deadlines are still satisfied. They provided a domain specific language for creating the monitors.

Jagadeesan and Viswanathan provide a formal discussion on observing properties in a system at run-time in [21]. They make a distinction between two kinds of run-time verification: active testing and passive testing. In the former, the observations of timed event traces are made from the initial state, while in the latter, observations of the system are obtained mid-stream. They identify that a property can be tested passively (i.e. it is acceptable to not observe the events all the time) if and only if it is prefix-closed and suffix-closed. The properties to be tested are modeled as a timed automaton. They provide an example system in which the passive properties are checked using UPPAAL [6] by checking if the composition of timed automaton generated by the observed trace and the property timed automaton is empty or not.

Goldberg and Horvath have discussed discrepancy monitoring [19] in the context of the health management architecture supported by ARINC-653. 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 Linear Temporal Logic (LTL) called MEDL in [39]. 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 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 the previous paragraph is that RT-MaC supports an "until" operator that allows the specification of a time bound where a given property must hold. Both of these works provided us with valuable input and influenced the design of our run-time component level health management.

Wang et al. [49] have described an online algorithm for checking past LTL properties of system execution. However, they allow uncertainty in observations by noting that the recorder might not capture the precise time the observations had occurred in the past.

9.2 Comparison-based anomaly detection

Comparison based detection schemes involve either executing the same component twice or executing at least three redundant units and then using a voting component. The comparison can be based on either exact agreement or approximate agreement, i.e. on all values being within a small distance of each other.

Comparison based detection schemes involve either executing the same component twice or executing at least three redundant units with a voting component. The comparison can be based on either exact agreement or approximate agreement, i.e. on all values being within a small distance of each other.

Laprie [23,24] describes different types of redundancy to detect different types of faults that can be captured by executing the software component twice. A point to note is that (binary) comparison based approaches can be used for detection but not masking. Moreover, they are susceptible to floating point precision errors. Theoretical analysis shows that at least three redundant units and a voter (3+1) are required to mask the detected fault. The categories listed below include the main variants, however, it must be noted that the overall approach appears to be more focused towards hardware faults.

Time-based redundancy: Execute the same software component twice but during two different time intervals. The main idea is that by comparing the two results separated by time, one can detect discrepancies caused by transient hardware faults that live for an interval shorter than the time elapsed between two executions. This approach will, however, lead to false positives when the component's output is dependent on the time of execution.

Hardware-based redundancy: Execute the same software component on two different hardware units. This can detect a transient or permanent hardware fault in any one of the units. It will not work when both hardware units fail in a similar way and cause the software to produce the same output.

Using diverse software on same hardware: Execute different versions of software on the same hardware across two different time intervals. This can catch faults that are introduced during implementation, although not common errors in the specification (discussed later). It can also detect transient hardware faults. This technique will produce false positives if the software output is time-dependent.

Using diverse software on diverse hardware: Execute different versions of software on independent hardware units. The comparison will detect faults due to design errors that generate different outputs in the different versions as well as any faults in the hardware.

9.3 Self-adaptive Software Systems

The work described here fits in the general area of self-adaptive software systems, for which a research road map has been presented in [10]. Our approach focuses on latent faults in software systems, follows a component-based architecture with a model-based development process and implements all steps in the collect-analyze-decide-act loop.

One notable approach to system health management for physical systems is to design a controller that inherently drives the system back into a safe region upon a system failure. This is the basis of the goal-based control paradigm [51] that supports a deductive controller responsible for observing the plant's state (mode estimation) and issuing commands to move the plant through a sequence of states that achieves the specified goal. This approach inherently provides fault recovery by using the control program to set an appropriate configuration goal that negates the problems caused by faults in the physical system. However, these control algorithms are themselves typically implemented in software and are therefore reliant on the fault-free behavior of related software components.

Conmy et al. presented a framework for certifying integrated modular avionics applications built on top of the ARINC-653 platform in [11]. 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 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 [33]. He identified that health monitoring is crucial for a safety-critical software system, and that in the future, it will be necessary to trade redundancy based fault tolerance for the ability of "reconfiguration on failure" while still operational. He described one possibility to achieve this goal using a set of lookup tables, similar to the health monitoring tables used in the ARINC-653 system specification, that map a trigger event to a set of system blueprints providing the mapping functions. Furthermore, he identified that this kind of reconfiguration is more amenable to failures that happen gradually, indicated by parameter deviations.

Rohr et al. advocate the use of architectural models for self-management [38]. They suggest the use of a run-time model to reflect the system state and provide reconfiguration functionality. From a development model, they generate a causal graph over various possible states of its architectural entities. At the core of their approach, they use specifications based on UML to define constraints, as well as monitoring and reconfiguration operations at development time.

Garlan et al. [18] and Dashofy et al. [12] have proposed an approach which bases system adaptation on architectural models representing the system as a composition of several components, their interconnections and properties of interest. Their work follows the theme of Rohr et al., where architectural models are used at run-time to track system state and make reconfiguration decisions using rule-based strategies.

While these works focus on the structural part of self-managing computing components, others have emphasized the need for behavioral modeling of the components. For example, Zhang et al. described an approach to specify the behavior of adaptable programs in [53]. Their approach is based on separating the adaptation behavior specification from the non-adaptive behavior specification in autonomic computing software. They model the source and target models for the program using Statecharts and then specify an adaptation model, i.e., the model for the adaptation set connecting the source model to the target model using a variant of Linear Temporal Logic [52].

Williams's research [37] concentrates on model-based autonomy. The paper suggests that an emphasis should be placed on developing techniques to enable software to recognize that it has failed and to recover from the failure. Their technique lies in the use of a Reactive Model-based Programming Language (RMPL)[50] for specifying both correct and faulty behavior of the software components. They also use high-level control programs [51] for guiding the system to the desirable behaviors.

Lately, the focus has started to shift towards formalizing concepts of self-management in software engineering. In [25], Lightstone suggested that systems should be made "just sufficiently" self-managing and should not have any unnecessary complicated functions. Shaw proposes a practical process control approach for autonomic systems in [42]. The author maintains that several dependability models commonly used in autonomic computing are impractical because they require precise specifications that are difficult to obtain. It is suggested that practical systems should use development models that include the variability and uncertainty inherent in the environment. Additionally, the development methods should not pursue absolute correctness regarding adaption, but instead should focus on the fitness for the intended task, or sufficient correctness. Several authors have also considered the application of traditional requirements engineering to the development of autonomic computing systems [7,45].

The work described here is closely related to the larger field of software fault tolerance: principles, methods, techniques and tools that ensure that a system can survive software defects that manifest themselves at run-time [27], [36]. Arguably, our approach comes closest to dynamic software fault removal, performed at run-time. The overall architecture presented below shows a specific implementation of the functions needed to perform this task.

10 Conclusion

This paper presented the design, implementation, and results of a deliberative, search-based strategy to restore the health of a software system. The three key design-time concepts are: (1) the system goals model, (2) the functional redundancy and allocation model, and (3) the implicit and explicit component operational requirements model. We described the semantics associated with each aspect of the individual models and used illustrative examples to derive the associated relationships. We presented key aspects of the runtime framework in detail, including the automatic generation of the runtime model from the static design-time specifications, the workings of the deliberative reasoner and the integration with external off-the-shelf solvers. The inherent workflow and the inter-relationship between the runtime elements was also described. Both Boolean and pseudo-Boolean satisfiability encodings for the reconfiguration problem were presented. Real-time, online-reconfiguration using the Deliberative Reasoner with an off-the-shelf Boolean Satisfiability Solver (CryptoMiniSat) was successfully demonstrated on both illustrative examples and a realistic case-study from the avionics domain.

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

References

1. Abdelwahed, S., Karsai, G., Mahadevan, N., Ofsthun, S.C.: Practical considerations in systems diagnosis using timed failure propagation graph models. Instrumentation and Measurement, IEEE Transactions on 58(2), 240–247 (February 2009)
2. ARINC: ARINC specification 653p1-3: Avionics application software standard interface part 1 - required services (2010), `https://www.arinc.com/`
3. Australian Transport Safety Bureau: In-flight upset; 240km NW Perth, WA; Boeing Co 777-200, 9M-MRG. Tech. rep. (August 2005), `http://www.atsb.gov.au/publications/investigation_reports/2005/aair/aair200503722.aspx`
4. Australian Transport Safety Bureau: AO-2008-070: In-flight upset, 154 km west of Learmonth, WA, 7 October 2008, VH-QPA, Airbus A330-303. Tech. rep. (October 2008), `http://www.atsb.gov.au/publications/investigation_reports/2008/aair/ao-2008-070.aspx`

5. Barry, M.: http://www.kestreltechnology.com/downloads/FailsafeOverview.pdf (2008)
6. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL - a tool suite for automatic verification of real-time systems. In: Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control. pp. 232–243. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1996)
7. Bustard, D.W., Sterritt, R.: A requirements engineering perspective on autonomic systems development. Autonomic Computing: Concepts, Infrastructure, and Applications pp. 19–33 (2006)
8. Butler, R.: A primer on architectural level fault tolerance. Tech. rep., NASA Scientific and Technical Information (STI) Program Office, Report No. NASA/TM-2008-215108 (2008), available at http://shemesh.larc.nasa.gov/fm/papers/Butler-TM-2008-215108-Primer-FT.pdf
9. Charette, R.: This car runs on code. IEEE Spectrum, February (2009)
10. Cheng, B.H.: Software engineering for self-adaptive systems. chap. Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26. Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-02161-9_1
11. Conmy, P., McDermid, J., Nicholson, M.: Safety analysis and certification of open distributed systems. In: International System Safety Conference,. Denver (2002)
12. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: Towards architecture-based self-healing systems. In: WOSS '02: Proceedings of the first workshop on Self-healing systems. pp. 21–26. ACM Press, New York, NY, USA (2002)
13. Dubey, A., Karsai, G., Mahadevan, N.: A component model for hard real-time systems: CCM with ARINC-653. Software: Practice and Experience 41(12), 1517–1550 (2011), http://dx.doi.org/10.1002/spe.1083
14. Dubey, A., Karsai, G., Mahadevan, N.: Model-based Software Health Management for Real-Time Systems. In: Aerospace Conference, 2011 IEEE. pp. 1–18. IEEE (2011)
15. Dubey, A., Karsai, G., Mahadevan, N.: Fault-adaptivity in hard real-time component based systems. In: de Lemos, R., Giese, H., Muller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. pp. 294–323. No. 7475 in Lecture Notes in Computer Science, Springer-Verlag (2013)
16. Dubey, A., Mahadevan, N., Karsai, G.: A deliberative reasoner for model-based software health management. In: The Eighth International Conference on Autonomic and Autonomous Systems (2012)
17. Eén, N., Sörensson, N.: An extensible sat-solver. In: Theory and Applications of Satisfiability Testing, 6th International Conference (SAT 2003). pp. 502–518 (2003)
18. Garlan, D., Cheng, S.W., Schmerl, B.: Architecting dependable systems. chap. Increasing system dependability through architecture-based self-repair, pp. 61–89. Springer-Verlag, Berlin, Heidelberg (2003), http://dl.acm.org/citation.cfm?id=1768179.1768183
19. Goldberg, A., Horvath, G.: Software fault protection with ARINC 653. In: Proc. IEEE Aerospace Conference. pp. 1–11 (March 2007)
20. Greenwell, W.S., Knight, J., 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. Jagadeesan, L.J., Viswanathan, R.: Passive mid-stream monitoring of real-time properties. In: EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software. pp. 343–352. ACM, New York, NY, USA (2005)
22. Johnson, S.B., Gormley, T.J., Kessler, S.S., Mott, C.D., Patterson-Hine, A., Reichard, K.M., Scandura, P.A.: System Health Management: With Aerospace Applications. John Wiley & Sons, Inc (2011)
23. Laprie, J.C.: Dependable computing and fault tolerance: Concepts and terminology. In: Proc. Twenty-Fifth International Symposium on Fault-Tolerant Computing, ' Highlights from Twenty-Five Years'. p. 2 (June 27–30 1995), http://ieeexplore.ieee.org/iel3/3846/11214/00532603.pdf?arnumber=532603
24. Laprie, J.C., Arlat, J., B'eounes, C., , Kanoun, K.: Architectural issues in software fault-tolerance. Software Fault Tolerance (1995), http://www.cse.cuhk.edu.hk/~lyu/book/sft/pdf/chap3.pdf, chapter 2
25. Lightstone, S.: Seven software engineering principles for autonomic computing development. ISSE 3(1), 71–74 (2007)
26. Lyu, M.R.: Software Fault Tolerance, vol. New York, NY, USA. John Wiley & Sons, Inc (1995), http://www.cse.cuhk.edu.hk/~lyu/book/sft/
27. Lyu, M.R.: Software reliability engineering: A roadmap. In: 2007 Future of Software Engineering. pp. 153–170. FOSE '07, IEEE Computer Society, Washington, DC, USA (2007), http://dx.doi.org/10.1109/FOSE.2007.24
28. Mahadevan, N., Dubey, A., Karsai, G.: Application of software health management techniques. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 1–10. SEAMS '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1988008.1988010
29. Mcintyre, M.D.W., Sebring, D.L.: Integrated fault-tolerant air data inertial reference system (1994)
30. Potocti de Montalk, J.: Computer software in civil aircraft. In: Digital Avionics Systems Conference, 1991. Proceedings., IEEE/AIAA 10th. pp. 324 –330 (oct 1991)
31. de Moura, L.M., Bjørner, N.: Z3: An efficient smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 337–340 (2008)

32. NASA: Report on the loss of the mars polar lander and deep space 2 missions. Tech. rep., NASA (2000), ftp://ftp.hq.nasa.gov/pub/pao/reports/2000/2000_mpl_report_1.pdf
33. Nicholson, M.: Health monitoring for reconfigurable integrated control systems. Constituents of Modern System safety Thinking. Proceedings of the Thirteenth Safety-critical Systems Symposium. 5, 149–162 (2007)
34. Ofsthun, S.: Integrated vehicle health management for aerospace platforms. Instrumentation Measurement Magazine, IEEE 5(3), 21 – 24 (Sep 2002)
35. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: A hard real-time runtime monitor. In: Runtime Verification. pp. 345–359. Springer (2010)
36. Pullum, L.L.: Software fault tolerance techniques and implementation. Artech House, Inc., Norwood, MA, USA (2001)
37. Robertson, P., Williams, B.: Automatic recovery from software failure. Commun. ACM 49(3), 41–47 (2006)
38. Rohr, M., Boskovic, M., Giesecke, S., Hasselbring, W.: Models in software engineering, workshops, and symposia at models 2006. In: Proceedings of the Workshop "Models@run.time" at the 9th International Conference on model Driven Engineering Languages and Systems (MoDELS/UML'06). vol. 4364 (2006)
39. Sammapun, U., Lee, I., 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. pp. 147–153 (17–19 Aug 2005)
40. Schumann, J., Srivastava, A.N., Mengshoel, O.J.: Who guards the guardians?: toward v&v of health management software. In: Proceedings of the First international conference on Runtime verification. pp. 399–404. RV'10, Springer-Verlag, Berlin, Heidelberg (2010), http://dl.acm.org/citation.cfm?id=1939399.1939432
41. Sha, L.: The complexity challenge in modern avionics software. In: National Workshop on Aviation Software Systems: Design for Certifiably Dependable Systems (2006)
42. Shaw, M.: "self-healing": softening precision to avoid brittleness: position paper for woss '02: workshop on self-healing systems. In: WOSS '02: Proceedings of the first workshop on Self-healing systems. pp. 111–114. ACM Press, New York, NY, USA (2002)
43. Sheffels, M.: A fault-tolerant air data/inertial reference unit. In: Digital Avionics Systems Conference, 1992. Proceedings., IEEE/AIAA 11th. pp. 127 –131 (Oct 1992)
44. Srivastava, A., Schumann, J.: The Case for Software Health Management. In: Fourth IEEE International Conference on Space Mission Challenges for Information Technology, 2011. SMC-IT 2011. pp. 3–9 (August 2011)
45. Taleb-Bendiab, A., Bustard, D.W., Sterritt, R., Laws, A.G., Keenan, F.: Model-based self-managing systems engineering. In: DEXA Workshops. pp. 155–159 (2005)
46. Torres-Pomales, W.: Software fault tolerance: A tutorial. Tech. rep., NASA (2000), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.8307, available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.8307
47. Tseitin, G.S.: On the complexity of derivations in the propositional calculus. Studies in Mathematics and Mathematical Logic Part II, 115–125 (1968)
48. Wang, N., Schmidt, D.C., O'Ryan, C.: Overview of the CORBA component model. In: Component-based software engineering: putting the pieces together, pp. 557–571. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
49. Wang, S., Ayoub, A., Sokolsky, O., Lee, I.: Runtime verification of traces under recording uncertainty. In: Proceedings of the Second international conference on Runtime verification. pp. 442–456. RV'11, Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-29860-8_35
50. Williams, B., Williams, B., Ingham, M., Chung, S., Elliott, P.: Model-based programming of intelligent embedded systems and robotic space explorers. Proceedings of the IEEE 91(1), 212–237 (2003)
51. Williams, B.C., Ingham, M., Chung, S., Elliott, P., Hofbaur, M., Sullivan, G.T.: Model-based programming of fault-aware systems. AI Magazine 24(4), 61–75 (2004)
52. Zhang, J., Cheng, B.H.C.: Specifying adaptation semantics. In: WADS '05: Proceedings of the 2005 workshop on Architecting dependable systems. pp. 1–7. ACM, New York, NY, USA (2005)
53. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: ICSE '06: Proceeding of the 28th international conference on Software engineering. pp. 371–380. ACM, New York, NY, USA (2006)