A Deliberative Reasoner for Model-Based Software Health Management

Abhishek Dubey, Nagabhushan Mahadevan, Gabor Karsai
Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN 37212, USA
{*dabhishe, nag, gabor*}*@isis.vanderbilt.edu*

***Abstract*—While traditional design-time and off-line approaches to testing and verification contribute significantly to improving and ensuring high dependability of software, they may not cover all possible fault scenarios that a system could encounter at runtime. Thus, runtime 'health management' of complex embedded software systems is needed to improve their dependability. Our approach to Software Health Management uses concepts from the field of 'Systems Health Management': detection, diagnosis and mitigation. In earlier work we had shown how to use a reactive mitigation strategy specified using a timed state machine model for system health manager. This paper describes the algorithm and key concepts for an alternative approach to system mitigation using a deliberative strategy, which relies on a function-allocation model to identify alternative component-assembly configurations that can restore the functions needed for the goals of the system. An example is used to show how such an approach can be used for performing automatic system reconfigurations, when faulty components are diagnosed.**

***Keywords-Component-based systems*; *fault diagnosis*; *autonomic computing*; *fault removal*.**

I. INTRODUCTION

Self-adaptive software systems, while in operation, must be able to adapt to latent faults in their implementation, in the computing and non-computing hardware; even if they appear simultaneously. Software Health Management (SHM) extends classical software fault tolerance techniques [1], [2], [3] by applying anomaly detection, fault source identification (diagnosis), fault effect mitigation (in operation), maintenance (offline), and fault prognostics (online or offline), as used in System Health Management of complex engineering systems [4], [5]. It is performed at run-time, and it includes fault detection, fault source isolation, and mitigation activities to remove fault effects. A good motivation for software health management is provided in [6].

We have developed an approach and 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 [7]. This framework brings the concept of temporal isolation, spatial isolation, strict deadlines from ARINC-653 and combines them with the well-defined component interaction patterns described in CORBA Component Model [8]. 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.

Higher-level management is provided by a System Health Manager (SLHM) that manages the health of the overall system. SLHM includes a diagnosis engine that uses a Timed Failure Propagation (TFPG) [9] model of the software that is automatically synthesized from the model of the software component assembly. The diagnostic engine reason about cascading fault effect 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 [10]. In the past, we showed how system-wide mitigation can be performed based on reactive timed state machines specified by the designer at the system integration time [11]. However, one of the problems with this approach to system-mitigation is the complexity of the specification required to cover all possible combination of failure scenarios.

This paper describes our work on system-level mitigation using a deliberative search-based technique that relies on the models of system goals/functionalities and the component groups allocated to provide these functionalities. Our approach is based on:

- Maintaining a model of the desired system functions and their sub-functions that all are required to provide that function.
- Maintaining an allocation tree for each function where the function is the root, the configuration groups (AND, OR, M of N) are the intermediate nodes and the software components are the leaf nodes. This tree captures the multi component configurations that are required to provide the service listed as the root function node.
- Identifying the current operational system goals.
- Identifying the affected operational goals based on the list of faulty components.
- Searching for alternative configuration that can satisfy the functions, while shutting down faulty components.

The outline of this paper is as follows: first we cover the related research. Then we present a short overview of our architecture and earlier results. Next, we discuss the deliberative reasoner, followed by a case study and conclusions.

 ISBN: 978-1-61208-187-8

II. Related Research

Our approach focuses on latent faults in software systems, it follows a component-based architecture with a model-based development process, and implements all steps in the Collect/Analyze/Decide/Act loop [12].

Rohr et al. advocate the use of architectural models for self-management [13]. 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. Garlan et al. [14] and Dashofy et al. [15] have proposed an approach which bases system adaptation on architectural models representing the system as a composition of several components, their interconnections, and properties of interest. They make reconfiguration decisions using rule-based strategies.

While these works have tended to the structural part of the self-managing computing components, some have emphasized the need for behavioral modeling of the components. For example, Zhang et al. described an approach to specify the behavior of adaptable programs in [16]. Their approach is based on separating the adaptation behavior specification from the non-adaptive behavior specification in autonomic computing software. Williams' research [17] concentrates on model-based autonomy. The paper suggests that emphasis should be on developing techniques to enable the software to recognize that it has failed and to recover from the failure. Their technique lies in the use of a Reactive Model-based Programming Language (RMPL)[18] for specifying both correct and faulty behavior of the software components. They also use high-level control programs [19] for guiding the system to the desirable behaviors.

Work described here is 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 [20], [21].

III. The ARINC Component Framework

System-level health management and fault tolerance approaches are based on the notion of interacting components. Our work is based upon the ARINC-653 component model (ACM) [7]. ACM combines the CORBA Component Model [8] with ARINC-653 [22]. ACM components interact with each other via well-defined patterns, facilitated by ports. In ACM, a component can have four kinds of ports for interactions: **publishers, consumers, facets** (a.k.a. provided interfaces - where an interface is a collection of related methods) and **receptacles** (a.k.a. required interfaces). The component can interact with other components through **synchronous** call/return interfaces (associated with facets or receptacles), and/or via **asynchronous** publish/subscribe event connections (between publisher and consumer). A component can also have internal methods that are periodically triggered. Systems are designed as composition of components using a modeling environment, which includes a domain specific modeling language and associated tools.

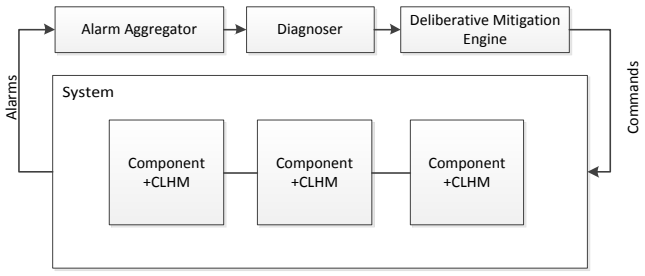

Figure 1. SLHM architecture.

Component Execution and Failure Scenarios: A software 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 ports of the component are operational. The active state of a component is the exact opposite of the inactive, and all the component ports are operational. In a semi-active state, only the consumer and receptacle ports of a component are operational, the publisher and provided ports are disabled. While the component is executing , i.e., it is in the active or semi-active state, the code in the component ports can introduce faults in the system, which can lead to anomalies in either the same component or in a connected component. We consider two root failure sources for each component port (a) a concurrency fault that is indicated by a timeout event in the act of obtaining the lock associated with the component, (b) or a latent bug in the code written by the developer to implement the operation associated with the port. Every component has a lock to ensure that at any given time at most one thread is active in the component.

Example: Figure 2 shows the assembly for a notional GPS system with a redundant set of Sensor and GPS components (Sensor2, GPS2). Deployment information is not shown in this figure. Sensors publish an event every 4 sec for their associated GPS. The GPS consumes the event published by its sensor at a periodic rate of 4 sec. Afterwards, it publishes an event, which is sporadically consumed by the Navigation Display. Thereafter, the display component updates its location by using getGPSData facet of the GPS Component. In the initial setup of the assembly, the Sensor, GPS, and NavDisplay components are used and hence set to be in *active mode*. The redundant sensor and GPS (Sensor2 , GPS2) are not used. The GPS2 is set to a *semi-active mode*, leaving the Sensor2 component in active mode. This would allow the GPS2 to keep track of the current state (by being in semi-active mode where the GPS2's consumers are active) but not affect NavDisplay.

 ISBN: 978-1-61208-187-8

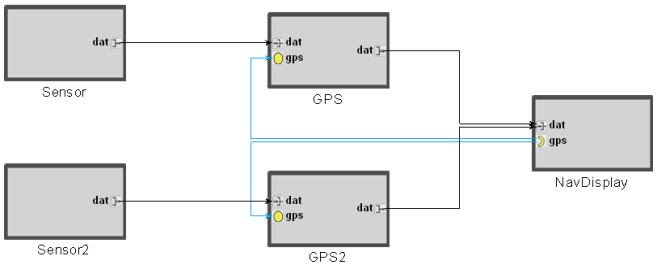

Figure 2. GPS Software Assembly.

A. Health Management in ACM

Health Management in ACM happens at two levels. The first level of protection is provided by a component level health management (CLHM) strategy, which is implemented in all components. It provides a localized timed state machine with state transitions triggered either by a local anomaly or by timeouts, and actions that perform the local mitigation. The System Level Health Manager (SLHM) is at the second, top level in our health management strategy. The deployment of the SLHM requires the addition of three special SLHM components to an ACM assembly: the *Alarm Aggregator*, The *Diagnosis Engine*, and the *Deliberative Mitigation Engine*, as shown in Figure 1.

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 collected using a moving window two hyperperiods long. The events are sorted based on their time of occurrence and then sent to the *Diagnosis Engine*.

The *Diagnosis Engine* hosts an instance of a Timed Failure Propagation Graph reasoner engine. This engine is initialized by a Timed Failure Propagation Graph (TFPG) [9] model that captures the failure-modes, discrepancies (possibly indicated by the alarms), and the failure propagations from failure modes to discrepancies and from discrepancies to other discrepancies, across the entire system [10], [11]. The reasoner uses this model to isolate the most plausible fault source: a software component that could explain the observations, i.e., the alarms triggered and the CLHM commands issued. The result, i.e., the list of faulty components is reported to the next component that provides the system level mitigation: the *Deliberative Engine*, discussed in the next section.

IV. Deliberative Engine

The mitigation engine in a system has to map the diagnosis results to a set of actions that remove the faults in the system and restore the functionality. There are four basic commands that can be sent to each component (a) RESET : Instructs a component to Reset itself, (b) STOP : Instructs a component to switch to inactive mode, (c) START: Instructs a component to switch to active mode, and (d) REWIRE

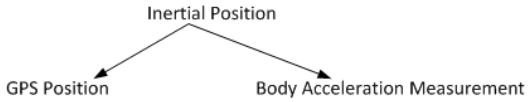

Figure 3. Example of Functional Decomposition for an Inertial Measurement Unit

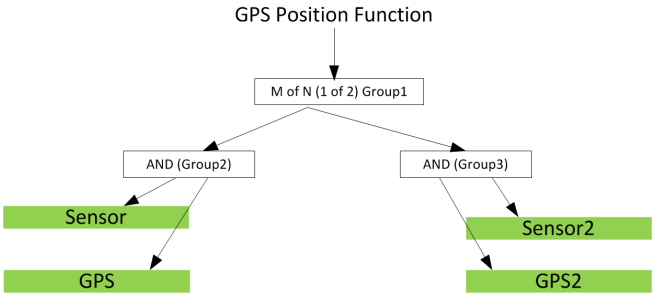

Figure 4. Example showing allocation of the GPS position function shown in Figure 3 to the components shown in assembly of Figure 1.

(ri,pc): Instructs a Component to rewire its receptacle Interface (ri) to connect to the appropriate facet interface in another component (pc). In case of REWIRE, the appropriate facet to be used is identified by the component which stores a map of component to facet for every receptacle in the component.

Our initial approach towards fault mitigation in SLHM included a *Reactive Mitigation Engine* wherein the mitigation strategy was specified as a hand-crafted, timed state machine model at design time. The updated fault status of the components in the assembly was used to trigger the SLHM state machine. For details on this mitigation specification, please see [10], [11].

The new SLHM mitigation approach uses a *Deliberative Mitigation Engine* which, like the reactive 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 deliberative mitigation approach relies on modeling the system goals as functions, the functional dependency on other sub-functions and the function-allocation model, i.e., the component group configurations that can provide the function (or sub-function). At run-time, the deliberative engine searches through the space of the function-allocation model to identify an alternate configuration of healthy components that can restore the functions (functionalities) affected by the faulty components.

In the timed-state machine approach, the modeler needs to specify the specific mitigation strategy for each fault (component) and/ or fault - combination (set of faulty components). We realized that the state-machine based approach (of modeling fault mitigation actions) is very tedious, error-prone and gets extremely complicated as the number of components and their fault combinations grow. In the current SLHM Mitigation strategy using Deliberative Reasoner, the reasoner builds a graph of the function allocation model

 ISBN: 978-1-61208-187-8

Table I
$IsUsable$ SEMANTICS

Type	Defintion
Component	$isUsable(c) \Leftrightarrow \neg isFaulty(c)$
AND-Group	$isUsable(g) \Leftrightarrow (\forall x \in child(g))(isUsable(x))$
XOR-Group	$isUsable(g) \Leftrightarrow (\exists x \in child(g))(isUsable(x))$
MofN-Group	$isUsable(g) \Leftrightarrow (\exists X \subseteq child(g))(\|X\| \geq M)$ $(\forall x \in X)(isUsable(x))$
Function	$isUsable(f) \Leftrightarrow (\forall x \in child(g))(isUsable(x))$

Table II
$isActive$ SEMANTICS.

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

and the assembly model and searches this graph for an appropriate mitigation action to restore the functionality. The deliberative reasoning approach using the function allocation model allows a better modeling scalability for faults and fault combinations.

Modeling the Functional Decomposition of System: During the design time, the system integrator enumerates the system functions as a collection of simple AND trees. That is, if $\mathbb{F}$ is the set of all immediate children of a function node, f_p, in the functional decomposition tree, then $isActive(f_p) = (\forall f \in \mathbb{F})(isActive(f))$.

Example Model: Figure 3 shows the functional decomposition of portions of an Inertial Measurement Unit. The Inertial Position function requires the GPSPosition function and the BodyAccelerationMeasurement function. In the running system one or more such function trees can be active. Additionally, a lower level function may be required in multiple trees.

Modeling the Functional Allocation: A function at any level of the functional decomposition directed acyclic graph can depend on other child functions and can depend upon the availability of a set of components at that level. The set of components related to a function can be hierarchically organized into groups. There are three kinds of groups: (a)**AND** of some components (all), (b) **XOR** of some components (exactly one of N), and (c) **MofN** of some components (at least M out of N components.). Note that both XOR and MofN groups are used to model redundancy.

Once specified, the functional allocation tree has the **function at the root, groups as intermediate nodes and components as the leaf nodes**. Components have two attributes in this tree: $isFaulty$ and $isActive$. While $isFaulty$ is determined based on the diagnoser output,

Procedure 1 Driver - RunDR

```
1: CLEAR LIST GRC, GRN.
2: for component c ∈ DR do
3:     MarkAsFaulty(c)
4: end for
5: RunReconfig();
6: return GRC; {set of possible reconfig commands}
```

$isActive$ is determined by the initial configuration. The deliberative reasoning process could result in marking a component (healthy or faulty) to be inactive , i.e., setting $isActive = false$. This results in sending a STOP command to the component. When a component is not faulty it is considered to be usable , i.e., $isUsable(c) \implies \neg isFaulty(c)$.

Usable attribute for the groups can be set based on the immediate child groups and child components. An AND group is usable if and only if all its children are usable. A XOR group is usable if any one of the children is usable. A MofN group is usable if at least M children are usable. These rules are summarized in Table I. Note in the table g means group, c means component. Operator $parent(x)$ returns the set of all immediate parents of x in the function allocation Directed Acyclic Graph (DAG). Operator $child(x)$ returns the set of immediate children of x, and $|.|$ is the cardinality operator.

Similarly $isActive(c)$ can be evaluated from leaf to the root of the function allocation tree. A root function in this tree is usable if all its immediate groups are usable. It is active if all its immediate groups are active. Table II summarizes all the rules. Note that due to the maximal nature of the $isActive(c)$ definition for MofN group, any reconfiguration action that requires turning a MofN group active requires to turn all its usable children active.

Example Function Allocation Model: Figure 4 shows the allocation diagram for one of the functions in figure components shown in Figure 3 using the components in the assembly depicted in Figure 2. The model indicates that the GPS Function can be provided by an M of N (1 of 2) Group. It requires the services of at least one of the two AND Groups (Group2 or Group3). The AND groups in turn require the services of all of their child nodes (here components).

A. Search and Reconfiguration Algorithms

During run-time, the deliberative engine is seeded with the functional allocation model translated into a XML form and the initial configuration of the system components. Internally, it maintains three lists: (a) Global List GFC: set of components that have been diagnosed as faulty, (b) Global List GRC: set of possible reconfiguration commands, and (c) Global List GRN: set of possible reconfiguration nodes.

Furthermore, the deliberative engine assigns an index to each node in the functional allocation graphs. All leaves are assigned index 0. Any other node has a level Number

 ISBN: 978-1-61208-187-8

Procedure 2 Mark as faulty

Input: Faulty Component c
Given: RN is an empty set.
1: **if** c ∈ GFC **then**
2: **return**
3: **end if**
4: c.isFaulty=**true**
5: c.isUsable=**false**
6: GFC.add(c)
7: RN = visitParent(c)
8: **if** isempty(RN) **then**
9: output-command = RESET(c)
10: **else**
11: output-command=STOP(c)
12: GRN.add(RN)
13: **end if**
14: GRC.add(output-command)

Procedure 3 visitParent

Input: Node N
Output: Set of Reconfig Node RN
1: $\mathbb{P} = parent(N)$
2: **for** $p \in \mathbb{P}$ **do**
3: **if** isUsable(p) **then**
4: RN.add(p) {add a usable node to possible reconfig nodes}
5: **return** RN
6: **else**
7: **if** $p \in GRN$ **then**
8: GRN.remove(p)
9: **end if**
10: **return** visitParent(p)
11: **end if**
12: **end for**
13: **return** 0

Procedure 4 RunReconfig

1: **for** n ∈ GRN **do**
2: Result= Reconfig(n)
3: **if** Result **then**
4: $CN = child(N)$ {set of children}
5: **for** ch∈ CN **do**
6: **if** $\neg isUsable(ch) \wedge isActive(ch)$ **then**
7: ReconfigStop(ch)
8: **end if**
9: **end for**
10: **end if**
11: **end for**
12: Check for Rewiring

that is Max(Level-Index of its children) +1. This Level-Index is used to sort the elements in GRN (the Possible-Reconfig Nodes) based on the graph topology. During the reconfiguration search, the elements in GRN are explored for reconfiguration in the increasing order of Level-Index , i.e., the reconfigurable nodes closest to the source is explored first.

Each time the deliberative reasoner is invoked, it receives an input list DR of components diagnosed as faulty. It invokes the steps detailed in Procedure 1. The deliberative reasoner uses the Procedure 2 to mark the faulty component in the functional allocation graphs. This algorithm does nothing if the component is already faulty. Otherwise, it marks it as faulty and invokes the visitParent Procedure 3. If the visitParent Procedure returns a possible reconfiguration node, then it records a command to STOP the faulty component. Otherwise, it records a command to RESET the faulty component. The reconfiguration node is added to the GRN, the command is added to the GRC.

The visitParent Procedure 3 is used to visit a parent node of the current node in the function allocation graph. It evaluates the IsUsable property for each parent node. If a parent node is still usable, then it returns the node as reconfiguration node. Otherwise, it recursively invokes VisitParent on the parent node. Note that each group has only one parent group or one or more function parent node.

Once the reconfiguration node, i.e., the node in the allocation tree where the change has to take place is identified, the Run Reconfig Procedure 4 is invoked to compute the reconfiguration that would restore the functionality. This algorithm loops through each node that is stored in the GRNlist. It invokes the Reconfig Procedure 5 on each node, which returns true if an alternative exists, else it returns false. It also invokes the ReconfigStop Procedure 6 on those child nodes that need to be stopped as they are no longer usable. Components that are marked as active but do not belong to any active parent group are commanded to be stopped. As a last step, it checks if any of the receptacles need to be rewired to a facet on a newly activated provider component. This step of rewiring is required if any component servicing a facet has been stopped in the current reconfiguration.

B. *Example Reconfiguration*

Consider the assembly captured in Figure 2. Initially Sensor, GPS, NavDisplay components are active. Sensor2 is also active. But, GPS2 is semi-active. Thus, GPS2 consumes data from the Sensor2 but does not publish data to NavDisplay. At this time, the Global List of Fault Candidates GFC is initialized as an empty list. The deliberative engine records the initial states of the component and identifies if the currently active functionality shown in fig 4 is satisfied or not.

The Deliberative Engine is invoked if there is any fault diagnosis reported by the Diagnosis Engine component. Consider that GPS component is reported faulty. This will lead to the invocation of the MarkAsFaulty Procedure 2, causing GPS to be set as faulty and unusable. When the VisitParent Procedure 3 is invoked, the parent group of GPS (And Group2) will be marked as unusable because it requires all children to be usable. A recursive call to the same Procedure will identify that the MofN Group 1 is still usable because at least 1 of the 2 AND groups, Group 3 is still usable. At the end of these two Procedures, Group1 will be added to the GRN and a command to stop the GPS

 ISBN: 978-1-61208-187-8

Procedure 5 Reconfig

```
Input: Node N
1: if isUsable(N) then
2:   if N.type() ==COMPONENT ¬isActive(N) then
3:     N.isActive= true
4:     Output-Command = START(N)
5:     GRC.add(Output-Command)
6:   end if
7:   if N.type() ==MofNGROUP then
8:     CN=child(N)
9:     for x ∈ CN do
10:      Reconfig(x)
11:    end for
12:    N.isActive= true
13:  end if
14:  if N.type() ==ANDGROUP then
15:    if isUsable(N) then
16:      CN=child(N)
17:      for x ∈ CN do
18:        Reconfig(x)
19:      end for
20:      N.isActive= true
21:    end if
22:    if N.type() ==XORGROUP then
23:      CN=child(N)
24:      for x ∈ CN do
25:        if isUsable(x) then
26:          Reconfig(x)
27:          for y ∈ CN and y ≠ x do
28:            if isActive (y) then
29:              ReconfigStop(y)
30:            end if
31:          end for
32:          N.isActive= true
33:          return {It will return as soon as the first is
             usable child is found}
34:        end if
35:      end for
36:    end if
37:  end if
38:  return
39: end if
```

component will be added to the GRC.

Once the reconfigurable nodes are identified, RunReconfig Procedure 4 will be invoked to identify the exact reconfiguration commands to restore the functionality. The Reconfig Procedure 5 will be performed on Group 1 which is of type MofN. This will result in iterative invocation of the Reconfig Procedure 5 on Group2 and Group3. While nothing will happen in the context of Group2 as it is no longer usable, Reconfig step will be invoked on Group3's children: Sensor2 and GPS2. Commands to START GPS2 component will be added to the Global Reconfig Command (GRC) list. The RunReconfig Procedure 4 will invoke ReconfigStop Procedure 6 on the other AND group (Group2) that will result in the GRC list being updated with a stop command for the Sensor component. An additional check will be performed to see if any receptacle ports need to be rewired. This results in the rewire of the receptacle in NavDisplay to use the facet in GPS2. Note the details of this check have not been included in the paper due to space constraints.

Procedure 6 ReconfigStop

```
Input: Node N
1: if N.type() ≠ COMPONENT then
2:   N.isActive=false
3:   CN=child(N)
4:   for x ∈ CN do
5:     ReconfigStop(x)
6:   end for
7: end if
8: if N.type() == COMPONENT then
9:   PN=parent(N)
10:  deactivate= true
11:  for x ∈ PN do
12:    if isActive(x) then
13:      deactivate = false
14:      BREAK
15:    end if
16:  end for
17:  if deactivate then
18:    output-command=STOP(N) {Deactivate a component,
       when none of its parents are active}
19:    GRC.add(output-command)
20:  end if
21: end if
```

C. Limitation

The algorithm described above suffers from a limitation exposed by the XOR group. The XOR group dictates that one and only one component associated with that group is active at any time. This condition associated with the XOR group could be violated in the algorithm described above. If one or more components appeared under an XOR group as well as in other branches of the function allocation model, the algorithm does not ensure that the XOR conditions are honored. Currently, we impose a restriction that a component featured under an XOR group may not feature elsewhere in the function allocation model.

V. Conclusion

This paper discussed our approach towards restoring the health of a software system based on identifying the alternative component configurations using the function-allocation model for the system. We described the design language for modeling the function allocation, the algorithm employed to update the usable branches of the function allocation model based on the fault report from the diagnosis engine and identify suitable component reconfigurations that can restore the function. An example was described to illustrate the function allocation design and the algorithm. In order to relax the restrictions associated with function allocation model, we are exploring other approaches such as using a SAT solver to identify alternate configurations.

Acknowledgement

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] Michael R. Lyu. *Software Fault Tolerance*, volume New York, NY, USA. John Wiley & Sons, Inc, 1995.

[2] Wilfredo Torres-Pomales. Software fault tolerance: A tutorial. Technical report, NASA, 2000. Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.8307.

[3] R.W. Butler. A primer on architectural level fault tolerance. Technical report, 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.

[4] S. Ofsthun. Integrated vehicle health management for aerospace platforms. *Instrumentation Measurement Magazine, IEEE*, 5(3):21 – 24, September 2002.

[5] S.B. Johnson, T. Gormley, S. Kessler, C. Mott, A. Patterson-Hine, K. Reichard, and P. Scandura Jr. *System Health Management: With Aerospace Applications*. John Wiley & Sons, Inc, 2011.

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

[7] Abhishek Dubey, Gabor Karsai, and Nagabhushan Mahadevan. A component model for hard real-time systems: CCM with ARINC-653. *Software: Practice and Experience*, 41(12):1517–1550, 2011.

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

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

[10] Abhishek Dubey, Gabor Karsai, and Nagabhushan Mahadevan. Model-based Software Health Management for Real-Time Systems. In *Aerospace Conference, 2011 IEEE*, pages 1–18. IEEE, 2011.

[11] Nagabhushan Mahadevan, Abhishek Dubey, and Gabor Karsai. Application of software health management techniques. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 1–10, New York, NY, USA, 2011. ACM.

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

[13] Matthias Rohr, Marko Boskovic, Simon Giesecke, and Wilhelm Hasselbring. *Models in Software Engineering, Workshops, and Symposia at MoDELS 2006*, volume 4364, chapter Model-driven Development of Self-managing Software Systems. 2006.

[14] David Garlan, Shang-Wen Cheng, and Bradley Schmerl. Architecting dependable systems. chapter Increasing system dependability through architecture-based self-repair, pages 61–89. Springer-Verlag, Berlin, Heidelberg, 2003.

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

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

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

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

[19] Brian C. Williams, Michel Ingham, Seung Chung, Paul Elliott, Michael Hofbaur, and Gregory T. Sullivan. Model-based programming of fault-aware systems. *AI Magazine*, 24(4):61–75, 2004.

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

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

[22] ARINC specification 653-2: Avionics application software standard interface part 1 - required services.

 ISBN: 978-1-61208-187-8