JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 8, April 2011

Autonomic Healing of Model-Based Systems

Steve Nordstrom, Abhishek Dubey* , Turker Keskinpala, Sandeep Neema, and Theodore Bapty
Vanderbilt University, Nashville, TN 37203

DOI: 10.2514/1.31940

Embedded computing is an area in which many of the self-* properties of autonomic systems are desirable. Model-based tools for designing embedded systems, while proven successful in many applications, are not yet applicable toward building autonomic or self-sustaining embedded systems. This is, in part, due to the domain-specific nature of modeling tools used to design and deploy embedded systems. The approach outlined in this paper uses a model-driven healing process to direct the system's adaptation in such a way that the resulting system configurations satisfy their particular domain-specific needs. A method for reducing the computational complexity of this healing process is also given.

I. Introduction

MISSION critical and safety critical systems require implementations that are resilient in the face of system failures. Embedded systems are being increasingly used to perform localized computation and control due in part to the decreasing size, increasing power efficiency, and increasing availability of embedded microcontrollers and their associated components. Many of these application scenarios require that the physical plants being controlled or monitored exhibit robust, fail-safe performance. Such examples include aircraft traffic control systems, nuclear power plant monitoring/control, flight avionics systems, space communications systems, and utility power production control systems. This increased usage of embedded computer systems for monitoring and controlling necessarily robust physical systems places a large demand on the need for reliable, fault-aware, and fault-tolerant embedded computer systems. Autonomic systems aim to provide this resiliency by adaptively mitigating present and potential faults. Significant design challenges arise when constructing a computer-based system capable of handling the uncertainty of multiple potential failures, occurring in arbitrary combinations and orders. Dynamically adapting the software of a running embedded system to meet new requirements often involves many factors which have to be understood by system designers in a very short time frame.

Data processing systems for high-energy physics particle accelerators continually push the limits of current processor, network, memory, and storage capacities. Traditional mechanisms for designing of fault-tolerant, maintainable, and efficient systems are not applicable to this task. When facing these realities, a group of physicists and engineers from both academia and US national laboratories formed the Real Time Embedded Systems (RTES) Group [1] to address reliability and design issues as they relate to large-scale fault-tolerant data processing systems. Such issues include the following:

1) The system must be highly available, because the system captures data continuously over a long period of time. Early estimations determine that for level of availability desired, a network of $\sim$2500 embedded processors in addition to another $\sim$2500 cluster machines is necessary.
2) To achieve this high availability, the system must be fault-tolerant, not using more than 10% of system resources for this objective.

Received 3 May 2007; accepted for publication 15 March 2009.
* dabhishe@isis.vanderbilt.edu.

3) Faults must be corrected in the shortest possible time and in both fully and semi-autonomous fashion (i.e., with as little human intervention as possible). Faults at the L1 [2] stage must be corrected on the order of 100 ms, whereas faults at the L2 operation may be corrected over a period of several seconds.
4) The system must be dynamically reconfigurable to allow a maximum amount of performance to be delivered from a set of indefinitely changing resources.

Determinability, that is, one's ability to determine the operating state(s) of the system, is of high importance in these systems. A growing number of embedded systems' design, implementation, and evolution rely heavily on the use of models. These systems are usually very tightly coupled to the modeling process in order to ensure that any guarantees made by analyzing the models are realized in the actual system. Designs are created, evolutions are made, and (re)deployments are all conducted from models. There is sometimes great difficulty in determining how system failures relate to the models, and how adaptations, redesigns, and evolutions of the systems should be performed.

When a system whose design is tied to models through a model-driven development process needs to adapt itself to a new environment, it becomes necessary that the system have internal knowledge of its own design (typically the same level of information captured in the models) so that it can make sense of how best to adapt. Conventional, model-driven methodologies focus on design, deployment, documentation, and reuse of software and models, and have made great strides in automating much of the software development process. However, even in model-driven development processes the design and evolution portions are very human-intensive. To achieve the goal of automated redesign of embedded systems, there is new ground that must be broken by bringing the concepts of autonomic computing together with embedded systems.

This paper presents a method for allowing model-based embedded systems to exhibit autonomic healing properties in order to solve the problems associated with online redesign of large-scale embedded systems. This is achieved through an integration of autonomic computing concepts with methods and practices of embedded system modeling methodology of model-integrated computing (MIC).

II. Modeling

Much work has been done to merge concepts of autonomy to scientific and embedded computing. Williams and the Model-based Embedded and Robotic Systems (MERS) Group's pioneering work in fault-aware systems details several desirable attributes of model-based and autonomous control systems [3,4]. In particular, an improved A* search technique that prunes conflicting states from the search space when investigating possible system adaptations in turn allows faster discovery of suitable adaptations [5]. Agarwal et al. [6] and Li and Parashar [7] address issues of design, development, composition, and deployment of grid applications using autonomic middleware services and components.

Garlan et al.'s work [8] in architecture-based self-repair details the use of architectural models to guide decisions regarding how best to adapt a system once failures are detected. The application of biologically based two-stage reflex–healing (RH) models as mechanisms for autonomicity and fault recovery in computer systems has been developed [9,10]. The application of MIC toward this problem has been outlined [11] and showed promise in this area because many design problems associated with autonomic and RH architectures could be alleviated with model-based techniques. For example, the use of integrating state machine-based modeling formalisms with application and deployment models to rapidly accommodate new reflexes has also been shown [11] and refinements toward verification of reflexes have been shown effective [12].

Software modeling has been gaining mainstream recognition for being a critical task in the process of designing tightly integrated software systems such as real-time and embedded systems. System and component properties and related information are captured and stored as models. Using these models, advanced tools can make greater sense of compositions of structures and associated interactions to provide many of the artifacts necessary to create a more reliable software product. Such artifacts can include (but are not limited to) timing simulations, control matrices, process schedules, additional source code, and configuration files. Models and advanced model-based tools use concepts of software and system modeling to describe components and interactions of a system and help to alleviate some of these concerns by enabling validation, code synthesis, and deployment assistance.

A. Model-Integrated Computing

In many cases, embedded design work is still extraordinarily complicated—much more so than typical software design. Examples of such cases include designing of large-scale high-performance systems or systems which operate in harsh or uncertain environments. Many of the struggles in designing these systems stem from the uncertainty of the future and of the environment. In such cases, the system's operational lifetime and component properties necessitate the expectance of component failures. As components fail in various ways, software can become unpredictable.

Model-integrated computing [13], which has been developed at the Institute for Software Integrated Systems at Vanderbilt University, is gaining acceptance in embedded system design and has shown great usefulness in modeling variety of simple and complex systems. The flagship software product that enables MIC is the Generic Modeling Environment (GME) [14]. Model-integrated computing allows designers to build domain-specific modeling languages and then use those languages to compose models of a system's objects and relationships. Model translators can then be used to extract useful information from the models. This information can be used for verification, simulation/analysis, code generation, and can also be in other areas of a software design and deployment processes.

Model-integrated computing has been shown to be an effective means of managing complexity in large-scale embedded systems [15] and is being shown to allow a growing variety of analyses to be performed on models [16–19].

B. Reflex and Healing Architectures

Management structures typically used to regulate large numbers of online processes or tasks are often hierarchical in nature and can therefore be visualized using tree structures. Our current RH architecture [16,20] is an example of one such architecture which uses a tree-like management structure to define the relationships between managers throughout a large-scale embedded system. The interactions between typical managers are shown in Figure 1.

This architecture supports four types of management interactions:

1) $S \leftrightarrow M$: interaction between an operator/system interface and a manager;
2) $M \overset{\text{up}}{\leftrightarrow} M$: interaction between a subordinate manager and a governing manager;
3) $M \overset{\text{down}}{\leftrightarrow} M$: interaction between a governing manager and subordinate managers;
4) $M \leftrightarrow T$: interaction between a manager and a managed task.

A Global Manager G is defined as a manager, which is capable of both $S \leftrightarrow M$ and $M \overset{\text{down}}{\leftrightarrow} M$ behavior. A Regional Manager R is defined as any manager that is capable of both $M \overset{\text{up}}{\leftrightarrow} M$ and $M \overset{\text{down}}{\leftrightarrow} M$ behaviors. Finally, local managers L are managers, which are only capable of $M \overset{\text{up}}{\leftrightarrow} M$ and $M \leftrightarrow T$ behaviors. Only local managers are capable of $M \leftrightarrow T$ interactions, and only global managers are capable of $S \leftrightarrow M$ interactions.

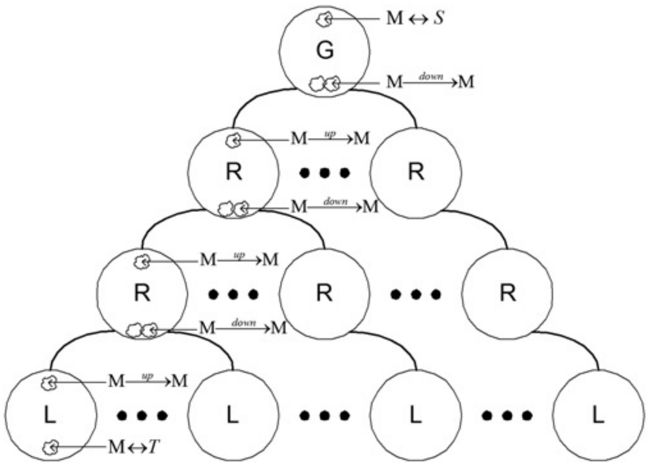

Fig. 1 Layout of management entities involving global (G), regional (R), and local (L) managers.

More challenging and still undiscovered are the aspects of model-based RH architectures that are associated with system healing after the initial reflexes have been enacted. Model-integrated computing places a considerable emphasis on information capture at design time and in the use of this information to synthesize a final set of system artifacts from an integrated model. If one were to examine the implementation of the resulting system, much of the information regarding the relationships between components is removed, because for any number of reasons (memory footprints, timeliness factors, and various optimization techniques) this information is no longer needed or deemed superfluous. However, in cases where a system must undergo redesign (such is the case for autonomic embedded systems), this information is of utmost importance.

Rather than attempting to design systems where this higher-level knowledge is pushed down into the running system, a cleaner solution is to integrate the modeling and design tools with the synthesized system by feeding information about the running system back into the modeling tools themselves. The modeling tools are coupled to the running system using a model synchronizer; this allows an existing model translator to be invoked on a model, which accurately represents the current state of the system. Work has been done to allow feedback from a live system to be included in the modeling tool [21]. Such feedback can be used to keep the model synchronized with the system; the authors recognize the tasks of fault diagnosis, and temporal model accuracy are non-trivial but are beyond the scope of this particular work.

III. Healing Through Model-Based Redesign

This investigation of complex information about a system is completely necessary in the case of adaptive systems, because there are a considerable number of events that could happen to necessitate an adaptation; perhaps far too many events to be handled by the system at a given time. What happens when the system needs to adapt to an environmental change but is limited by the events it can handle? Designers work very diligently to rule out such cases, but they are not unavoidable.

In the case where the system is unable to adapt to its new environment, the designer must revisit the models using the modeling tools, add the necessary event handling, fault scenarios, or fault mitigation rules, and then redeploy the system. Clearly, this can be done, but at a cost; the designer must have the knowledge and time to perform the necessary modifications and the system must be in a state where it can wait for the necessary modifications. Over time, designers oscillate between the design and re-deployment cycle after the system has been initially deployed. In the pursuit of autonomicity, it becomes desirable to have modeling tools which are capable of supporting a more automated redesign process. The following guidelines are put forth to bind a solution for autonomic embedded systems to a set of criteria. The autonomic redesign process must:

1) require minimal human interaction, as subject to the guidelines of autonomic computing [22];
2) retain the benefits of MIC, using the same model formats and model transformations available to a designer executing a manual redesign;
3) support a variety of model healing strategies;
4) retain the system's ability to perform healing operations in the future; and
5) include the ability to accommodate human-in-the-loop control of healing (allowing a human to evaluate the decisions of the redesign process before changes to the system are enacted).

In order that one might automate the process of redesign, it is necessary to understand the manual redesign process to a degree that one can automate it in software. To do this, we must attempt to describe the design effort in a way that makes sense algorithmically.

A. Arriving at a Healed Model

The term "healer" is used to describe the model transformation engine, which performs the task of finding a new model that is most suited to operate in the new environment. A designer has a limited set of operations that can be applied to the model. We will consider only the operations that lead to healing. (There are many non-essential operations a designer can perform using a modeling tool; the changing of a model's color, name, or other trivial operations that do not lead to healing are not considered by the automated healing tool.)

To arrive at a suitable model, the healer will first produce a set of candidate models in accordance with a set of healing actions allowed. A model is considered healed when it passes a testing function to determine if any faults are still present in the model. After one round of healing, a number of the resulting models may be considered healed,

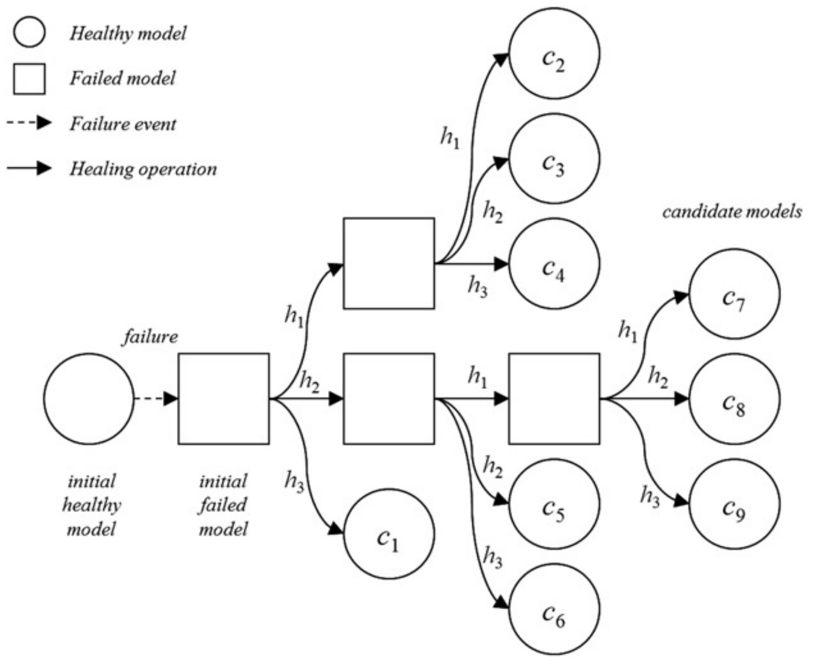

Fig. 2 Determination of candidate models from an initially failed model through healing action sequences.

whereas a number may not. The procedure will continue until all the candidate models are healed. Figure 2 shows this process for a failed model with three possible healing operations.

B. Choosing the Best Candidate Model

Once the set of candidate models is formed, the healer must then chose which of the set is the most appropriate to be used on the redesign. It has been shown that this process is multi-objective in nature and that the process is dependent on the factors driving the evaluation criteria [23]. Some examples of suitable evaluation criteria include 1) raw predicted performance of the model with respect to its data processing capability (number of packets processed per unit time); 2) the cost to migrate the system from the existing state toward compliance with candidate healed model, or 3) the model's suitability to handle future failures.

C. Special Criteria of Resilience

As proponents of fault-tolerant design, we would prefer to study more closely the process of finding a model's suitability to endure future faults. We use the term "resilience" to describe a measure of the candidate model's ability to withstand a single-component failure at some future time given a set of possible failures and evaluation criteria.

The measure of resilience is made by applying all possible failures for a candidate model to arrive at a set of possible failure states. For each of these states, the healer can perform the healing operation to form the next generation of candidate models. These models are then evaluated for their suitability. One can see that this process may be unending, and so a determination must be made as to how far into the future the healer can look to determine the next healed model; this is referred to as the look-ahead depth of the healing process. For this last stage, the healer omits the resilience criteria and evaluates with remaining models with no further failure/healing propagation. A general illustration of this process is given in Fig. 3. Portions of work have been done in this area [24]; however, more clarification is needed about how a designer adapts an existing software model to accommodate change in the system.

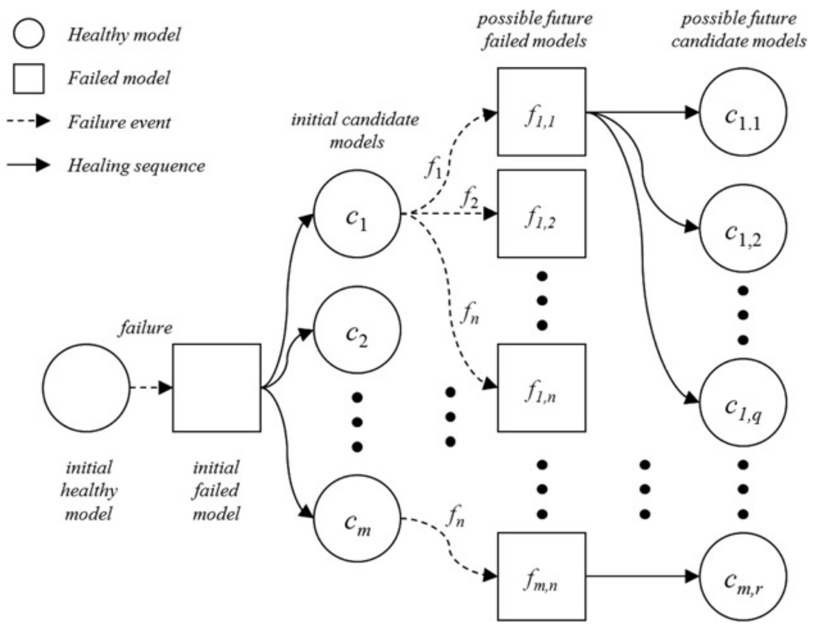

Fig. 3 Determination of the most resilient model involves applying potential failures to candidate models and evaluating the resulting set of healed models.

IV. Tools for Model-Based Healing

Building on our previous work, we present our modeling framework and toolset, which is progressing toward a complete MIC toolset for designing, building, and deploying autonomic embedded systems. The tools consist of two major components: a domain-specific modeling language for autonomic embedded systems and a healing model translator to determine how best to redesign the system in the event of component failures and automatically produce the healed model.

At the heart of MIC is the creation of graphical languages which are specific to a given problem domain. Model-integrated computing relies heavily on the use of these domain-specific modeling languages (DSMLs) to capture relevant characteristics of an object or system of objects. A DSML allows a designer to describe objects in terms of the domain rather than in terms of traditional computer languages. The generic modeling environment (GME) [14]† is a freely available and open source tool that provides a platform upon which MIC design and development is performed. Succinctly, the GME is a configurable and domain-independent modeling environment that supports the creation and instantiation of multiple user-defined (domain-specific) modeling languages.

A. Domain-Specific Modeling Language

First, the Guided Healing and Optimization Search Technique Modeling Language (GHOSTML) [24] is a DSML created for specifying components and interactions of large-scale embedded systems. This is done by using three distinct aspects in which the components of the system are modeled. The aspects are the following:

1) *Tasks*: A hierarchy of management tasks is created in the Tasks aspect, which determines the structure of the RH hierarchy. Containment and hierarchical decomposition of tasks are both modeling techniques that are used to manage complexity in the Tasks aspect (e.g., managed tasks are contained inside their governing manager).
2) *Networking*: Computational resources (nodes) as well as data visualization and interconnect resources (routers) are modeled in the networking aspect.
3) *Allocation*: The mapping of tasks onto assigned resources is done through associations modeled in the allocation aspect. Tasks are mapped onto resources in a many-to-one fashion.

† The ESCHER Research Institute. http://www.escherinstitute.org.

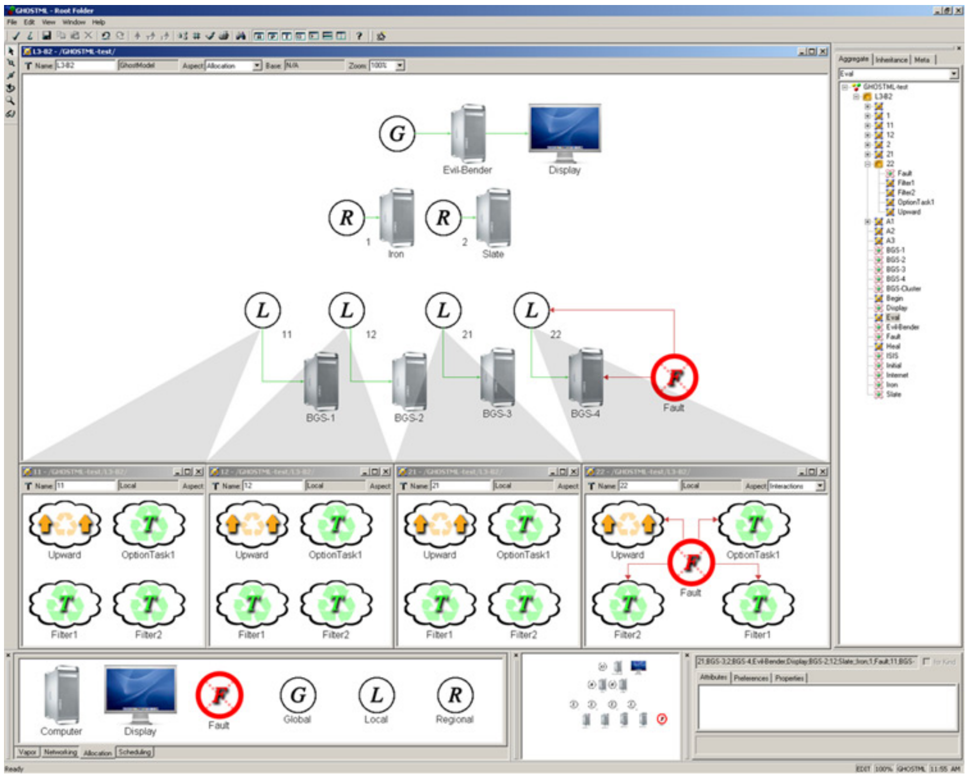

Fig. 4 Modeling tools allowing the capture of components in both operational and failed states, showing several nodes and their associated managers, one of which has failed.

Models expressed in GHOSTML can be used to indicate the presence of failures in the system. This is done through the use of a Fault object. All aspects of GHOSTML allow the specification of a Fault object. Fault objects can be associated with any component of the GHOSTML language (through containment or through an `isFaulty` association). Figure 4 shows the view of the Allocation aspect in GME. As GHOSTML task models also specify the set of reflex behaviors inside each component of the management hierarchy, the presence of faulty reflex actions can also be modeled. However, the work described in this paper deals primarily with the structural aspects of GHOSTML models. Models which contain `isFaulty` associations and/or Fault objects are said to be Failed models. It is assumed that the faulty/nominal state of each component in the system is accurately reflected in the model when a healing action must be performed.

B. The Healer: An Autonomic Model Translator

Secondly, a specialized model transformation tool called a Healer is used to perform the action of healing a failed model. The Healer uses a technique detailed in Sec. III that explores all possible healing actions, which produce a candidate healed system model. These candidates are then evaluated against a set of performance criteria, one of which is the resilience criteria, also described in Sec. III. To determine the resilience of a given candidate model, it is transformed in all possible ways into a failed model (using only single failures), and each failed model is subjected to further healing and evaluation. Pseudo-code for healing and failing a model during the search are shown below.

```
function Heal(Model m):
H := set of all healing operations
max := Empty{Model, Xfm}
Mh := {Empty{Model,Xfm}}
depth := depth + 1
```

```
for all h in H
   mc := ApplyXfm(h, m)
   if (depth <= maxdepth)
      Mf := Fail(mc)
         for all mf in Mf
            if (Eval(Heal(mf)) > Eval(max))
               max := mc
   else
      if (Eval(mc) > Eval(max))
         max := mc
return max;
function Fail(Model m):
   F := set of all possible 1-node failures
   Mf := Empty{};
   for all f in F
      append ApplyXfm(m, f) to Mf
return Mf
```

In addition to the `Heal()` functionality listed above, the healer is appointed with a set of healing operators that perform a primitive set of basic, well-defined operations on a GHOST model, the detail of which are as follows:

Kill (Manager): Removes the selected Manager from the active model and places it in a storage area for bookkeeping purposes. Any Allocations for which the Manager was a participant are removed, and any Tasks the Manager is responsible for are Killed.

Kill (Resource): Removes the selected Resource from consideration in the active model. Any Allocations for which the Resource was a participant in are removed. However, any connections to other Resources are maintained.

Kill (Task): Removes the selected Task from the active model and places it in a storage area. The governing Manager of the selected Task is no longer responsible for that Task. Any information about the Manager of the task is not preserved. The Task's Manager is otherwise not affected.

Revive (Manager): Removes the selected Manager from the storage area and places it back in the active model. No Allocations in which the Manager was previously participating in are recreated, nor are any Tasks the Manager was previously responsible for Revived.

Revive (Resource): Places the selected Resource back into consideration within the active model. Any Allocations in which the Resource was participating in are not automatically restored.

Revive (Task, Manager): Replaces the selected Task from storage under the control of the selected Manager.

Clone (Task, Manager): Duplicates an instance of the selected Task and places it under the control of the selected Manager.

Reallocate (Manager, Computer): Release any current Allocation from the selected Manager, and allocates it to the selected Computer.

PromoteManager (Manager): The selected Manager will assume the role of his governing Manager (Local becomes Regional, Regional becomes Global, etc.). The selected Manager is placed under the control of his governing Manager's governing Manager. The selected Manager's governing Manager's Tasks are Cloned to the selected Manager and the old governing Manager is Killed. No Allocation interactions are changed. A subordinate Manager under the selected Manager is chosen for Promotion. This is a recursive operation.

BecomeSubordinate (ManagerSub, ManagerGov): Releases ManagerSub's governing Manager from his responsibility of ManagerSub. Places ManagerSub under the responsibility of ManagerGov.

Recall that the best choice candidate is the model that provides the highest degree of satisfaction in the evaluation criteria. For the criteria of resilience, the best choice candidate is the model whose descendant failed models after healing show the best suitability with respect to the evaluation criteria. The healer proceeds by conducting an adversarial search game using two players [25] to explore portions of the game's search space. The players are Heal and Fail, and alternate turns building a state space game tree similar to the tree shown in Fig. 3. The game proceeds in a minimax-like fashion [26] using a multi-objective heuristic to evaluate the utility of each node of the tree. As the game is too complex to search to completion, a depth cutoff (also referred to as the number of plies) is used to limit the scope of the look-ahead. The value of this cutoff is dependent on the move set (which therefore limits to the branching factor of the search tree) and the computational resources required by the Healer.

As in other deterministic game searches such as chess or checkers, full knowledge about the game can be observed by both players, and the moves allowed by each player is known. Each player is allowed to make a single legal move (in reality, the healing operation can be constructed as a composition of atomic operations but for simplicity we shall consider as single moves). The Heal player's allowable moves consist of the set of model transformation chains which, when applied to the current model, result in a model containing no faults, as discussed in Sec. III. The Fail player's set of allowable moves consist of all model transformations which introduce single failure associated with any component in the model.

In reality, the game is being played while the embedded system is running, the Heal player acting as the healer, and the Fail player acting as the uncertainty in the environment. The situation is similar to that of a chess program playing a human opponent in that the set of moves allowed by the human is known so that the computer is able to compute its best move given the rules of the game but it must wait for the human to make move before it can proceed with a new search [27,28]. In the same way, the Healer must wait for a failure to occur before it can calculate its best move to heal the system. This is similar to a game against nature [29] in which a uniform random variable is used to predict the moves of a disinterested opponent.

Once a move is chosen by the Healer, it is applied to the model where the tools then use a special translator to implement the healing of the failed system. For the time being, it is assumed that the model on which the game is based is an accurate reflection of the currently running system for the duration of the healer's turn. It is assumed that the rate of failures occurring in the environment is sufficiently slow to allow a search to be conducted (as limited of depth as it may be) before the next failure occurs.

There is a problem, however; failing each component in the candidate model places a considerable strain on the computation time of the game. To alleviate this condition, one can form an equivalent but simplified model which greatly reduces the number of failed models produced by the Fail function, while still retaining the notion that all classes of failures will be enumerated, thereby reducing the number of resulting failed models which need to be healed and evaluated for resilience.

Given two nodes $n_1, n_2 \in N$, where N is the set of all nodes in a GHOSTML model, one can define a relation $\sim_s$ such that $n_1 \sim_s n_2$ if and only if the following conditions are satisfied:

1) the number of Managers allocated on n_1 is equal to the number of Managers allocated on n_2;
2) the number of Tasks residing on n_1 is equal to the number of Tasks residing on n_2; and
3) the specific instances of Tasks residing on n_1 are the same as those residing on n_2.

From these conditions, we can observe the following:

$$n_1 \sim_s n_1 \implies \sim_s \text{ is reflexive} \tag{1}$$

$$n_1 \sim_s n_2 \implies n_2 \sim_s n_1 \implies \sim_s \text{ is symmetric} \tag{2}$$

$$n_1 \sim_s n_2 \text{ and } n_2 \sim_s n_3 \implies n_1 \sim_s n_3 \implies \sim_s \text{ is transitive} \tag{3}$$

Because $\sim_s$ is reflexive, symmetric, and transitive, we can say that $\sim_s$ is a similarity relation, and we can form equivalence classes denoted as $[n_i]$ such that

$$\bigcup_{i:1..r} [n_i] = N \tag{4}$$

A reduction to a candidate model can now be made using this relation. First, the healer generates a set of candidate models. Then, it applies the similarity relation to the nodes of each candidate to annihilate all nodes which are similar to a given node. The fail player then builds its move set (possible faults) for each reduced candidate model. The game branches by applying each fault from the reduced model to the non-reduced model candidate, and the game continues. The result is an overall decrease in the number of faults the fail player could place within the model, which results in a smaller search tree.

As an example, consider the simple and ideal RH structure as shown in Fig. 5. Although the addition of the node's computational and network performance can easily be added to the equivalence relation above, for the sake

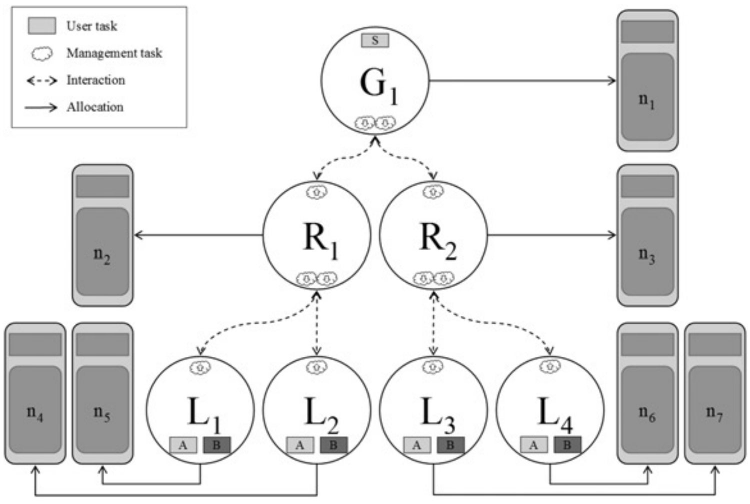

Fig. 5 An example GHOSTML model showing elements of the RH structure mapped to unique and identically specified nodes.

of simplicity we assume that each manager is deployed on identically capable nodes. To evaluate the resilience of this structure, the algorithm must fail the node in every possible manner. If the Healer were to naively fail every node in this model, the resulting set of failed models would contain seven models, which would in turn need to be healed and evaluated.

However, by constructing equivalence classes out of the nodes in the structure (as seen in Fig. 6), we can see that the number of components has been reduced. The model shows a single representative node of each equivalence class. The set of failed models produced by the reduction algorithm are reduced from seven in the full model to three in the reduced model. One will notice that the number of failed models produced by the Fail function is exactly the number of equivalence classes formed from the original candidate model. The effect of this reduction becomes more pronounced as the number of nodes in the model increases. Subsequent failures, however, may increase the number of equivalence classes, but even in the worst case, by definition the number of equivalence classes will be less than or equal to the total number of nodes. The worst scenario will happen only if there exists no similarity between any of the nodes; essentially the system would be completely heterogeneous. This technique therefore rewards homogenization of the RH structure and hardware resources.

One can see that from the original model the following equivalence classes are created:

$$\begin{aligned} &[n_1] = \{n_1\} \text{ since } \forall n \in N/n_1 \neg(n_1 \sim_s n) \\ &[n_2] = \{n_2, n_3\} \text{ since } n_2 \sim_s n_3 \\ &[n_4] = \{n_4, n_5, n_6, n_7\} \text{ since } n_4 \sim_s n_5 \sim_s n_6 \sim_s n_7 \end{aligned} \tag{5}$$

C. Correctness of the Healer

To absolutely declare that a healer is functioning properly, two criteria must be satisfied:

1) Given any input model, the healer must never produce a candidate model which is syntactically incorrect, that is, it always produces "valid" models.
2) All models produced by the healer must be considered "correct".

To satisfy the first criterion for all cases, the language used to express the healing strategies must be amenable a formal verification process. Additionally, the second criterion can only be satisfied if one can ensure that for any valid GHOSTML model the behavior of the exemplified system is also considered "correct." This is difficult because one must incorporate a model of the target system in order to rigorously verify properties of the model and correlate them to properties of the actual system. There is also the added concern of defining the "correctness" boundaries for the represented system.

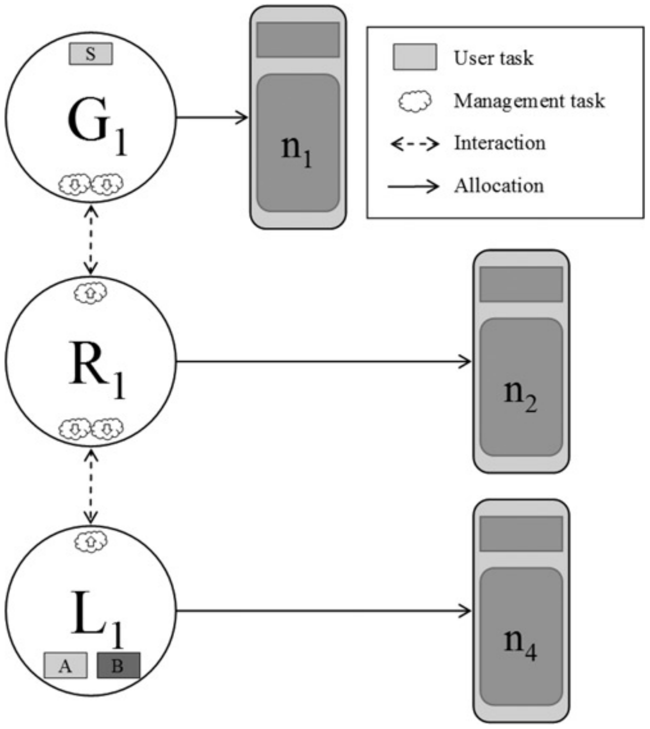

Fig. 6 Reduction of model to be failed using equivalence classes, one representative pairing of resources and tasks is shown for each class.

To avoid the difficulties and pitfalls associated with model verification and verifiable languages, a more relaxed definition of healer correctness must be considered. Heuristic methods can be used to check instances of models produced by the healer both for syntactic correctness and fitness of the resulting system. Additional methods can be used to compare the two output models to determine which is “more correct.” The strategy for this is to choose an alternate representation for the candidate model and try to correlate some well-defined properties of that alternate model with desired properties of the candidate.

For example, one can very easily represent a GHOST model as a simple directed graph and then correlate well-known graph examination algorithms to desired properties of the model to make comparative statements about GHOST models. For example, one can choose to favor a model whose network graph has the highest cut set, as this means there exist the most links that could fail without isolating any single resource. Also, one might favor a management tree with a desired levels of manager hierarchy by applying the Floyd–Warshall algorithm [30], which determines the minimum distance path between all node pairs in a graph. The idea is to construct relations between the desired properties of a candidate model and the results of well-known algorithms applied to an alternate representation of the model.

V. Conclusions and Future Work

Certainly the process of finding an appropriate healed model for a given failed model can be a difficult decision. There may exist a large set of evaluation criteria as well as healing strategies to be considered, and different designers may choose different healed models. This solution can then be used to automate the process of healing a failed model. As embedded systems grow more pervasive, distributed, and autonomic, the advancement tools such as these are going to have a significant impact in the way embedded software systems are designed, built, and maintained. Our work will progress in the areas of distributed computation of and evaluation of candidate models in embedded supercomputing environments, as well as in a scientific evaluation of how far into the future a healer can look for a given problem and still arrive at a meaningful solution in acceptable time.

Future efforts in this area will include the ability to define probabilities for certain types of faults, as the resilience evaluation currently presented treats all faults as equally likely. This leads to an adversarial search problem in which the fail player is motivated. A motivated fail player will likely result in the need to move toward a more sophisticated probabilistic search algorithm such as Expectimax [25]. Another avenue of exploration in this realm includes the use of pruning techniques based on the evaluation of candidates at a given level of ply during the enumeration of the search tree. Techniques such as alpha–beta pruning could be further used to eliminate low-scoring nodes of a game search tree without the need to create branches to explore possible failures of those nodes. The combination of equivalence classes with alpha–beta pruning to reduce search trees might serve to further reduce search complexity as the system model tends toward complete heterogeneity over time.

The ability to modify reflex behaviors during a healing transformation would also be beneficial, as a reduction in managerial capability might help alleviate the strain of running under a reduced set of hardware. However, this has some far-reaching implications with respect to model transformations which explicitly change a system's behavior; a more stringent means of evaluating candidate models with regard to overall system behavior needs to be developed.

Acknowledgments

This work was funded by the National Science Foundation ITR grant ACI-0121658. The authors also acknowledge the contribution of other team members of the Real-Time Embedded Systems (RTES) Group at Fermi National Accelerator Lab, University of Illinois at Urbana-Champaign, University of Pittsburgh, and Syracuse University.

References

[1] The Real Time Embedded Systems Group. http://www-btev.fnal.gov/public/hep/detector/rtes [retrieved 14 March 2007].

[2] Votava, M., "Btev Trigger/daq Innovations," *14th IEEE-NPSS Real Time Conference*, IEEE Computer Society, Stockholm, Sweden, 2005, 4–10 June 2005, p. 5.

[3] The Model-based Embedded and Robotic Systems Group at the Massachusetts Institute of Technology. http://mers.csail.mit.edu/mers.htm.

[4] Williams, B. C., Ingham, M., Chung, S., Elliott, P., Hofbaur, M., and Sullivan, G. T., "Model-based Programming of Fault-Aware Systems," *AI Magazine*, Vol. 24, No. 4, 2004, pp. 61–75.

[5] Williams, B. C., and Ragno, R., "Conict-Directed a* and Its Role in Model-based Embedded Systems," *Journal of Discrete Applied Math, Special Issue on Theory and Applications of Satisfiability Testing*, Vol. 155, No. 12, January 2003, pp. 1562–1595.

[6] Agarwal, M., Bhat, V., Li, Z., Liu, H., Khargharia, B., Matossian, V., et al., "Automate: Enabling Autonomic Applications on the Grid," *Proceedings of the 5th Annual International Active Middleware Services Workshop on Autonomic Computing* (AMS2003), IEEE Computer Society, Seattle, WA, USA, June 2003, pp. 48–57.

[7] Li, Z., and Parashar, M., "Rudder: A Rule-based Multi-Agent Infrastructure for Supporting Autonomic Grid Applications," *Proceedings of the First International Conference on Autonomic Computing*, IEEE Computer Society, Washington, DC, USA, 2004, pp. 278–279.

[8] Garlan, D., Cheng, S. W., and Schmerl, B., "Increasing System Dependability Through Architecture-based Self-Repair," *Architecting Dependable Systems*, Vol. 2677, 2003, pp. 61–89.

[9] Sterritt, R., Gunning, D., Meban, A., and Henning, P., "Exploring Autonomic Options in An Unified Fault Management Architecture through Reflex-Reactions via Pulse Monitoring," *Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-based Systems*, IEEE Computer Society, Washington, DC, USA, 2004, 24–27 May 2004, pp. 449–455.

[10] Sterritt, R., and Bantz, D. F., "Personal Autonomic Computing Reflex Reactions and Self-Healing. Systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, Vol. 36, No. 3, May 2006, pp. 304–314. doi: 10.1109/TSMCC.2006.871592

[11] Shetty, S., Nordstrom, S., Ahuja, S., Yao, D., Bapty, T., and Neema, S., "Systems Integration of Large-Scale Autonomic Systems using Multiple Domain-Specific Modeling Languages," *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems* (ECBS'05), IEEE Computer Society, Washington, DC, USA, 4–7 April 2005, pp. 481–489.

[12] Dubey, A., Nordstrom, S., Keskinpala, T., Neema, S., Bapty, T., and Karsai, G., "Towards a Verifiable Real-Time, Autonomic, Fault Mitigation Framework for Large Scale Real-Time Systems," *Innovations in Systems and Software Engineering*, Vol. 3, No. 1, 2006, pp. 33–52.

[13] Sztipanovits, J., and Karsai, G., "Model-Integrated Computing," *Computer*, Vol. 30, No. 4, 1997, pp. 110–111. doi: 10.1109/2.585163

[14] Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., et al., "The Generic Modeling Environment," *Workshop on Intelligent Signal Processing*, Budapest, Hungary, Vol. 17, May 2001.

[15] Ahuja, S., Bapty, T., Cheung, H., Haney, M., Kalbarczyk, Z., Khanna, A., et al., "Rtes Demo System 2004," *SIGBED Rev.*, Vol. 2, No. 3, 2005, pp. 1–6.
doi: 10.1145/1121802.1121804

[16] Neema, S., Bapty, T., Shetty, S., and Nordstrom, S., "Developing Autonomic Fault Mitigation Systems," *Journal of Engineering Applications of Artificial Intelligence Special Issue on Autonomic Computing and Grids*, Vol. 17, No. 7, 2004, pp. 711–725.

[17] Szemethy, T., and Karsai, G., "Platform Modeling and Model Transformations for Analysis," *Journal of Universal Computer Science*, Vol. 10, No. 10, 2004, pp. 1383–1407.

[18] Madl, G., Abdelwahed, S., and Schmidt, D. C., "Verifying Distributed Real-Time Properties of Embedded Systems via Graph Transformations and Model Checking," *Real-Time Systems*, Vol. 33, Nos. 1–3, 2006, pp. 77–100.
doi: 10.1007/s11241-006-6883-y

[19] Manders, E.-J., Biswas, G., Mahadevan, N., and Karsai, G., "Component-oriented Modeling of Hybrid Dynamic Systems using the Generic Modeling Environment," *Fourth and Third International Workshop on Model-Based Development of Computer-Based Systems and Model-Based Methodologies for Pervasive and Embedded Software* (MBD/MOMPES 2006), IEEE Computer Society, Potsdam, Germany, 30 March 2006, p. 10.

[20] Nordstrom, S. G., Shetty, S., Neema, S., and Bapty, T., "Modeling Reflex-Healing Autonomy for Large-Scale Embedded Systems," *IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews*, Vol. 36, No. 3, 2006, pp. 292–303.
doi: 10.1109/TSMCC.2006.871597

[21] Messie, D., Jung, M., Oh, J. C., Shetty, S., Nordstrom, S., and Haney, M., "Prototype of Fault Adaptive Embedded Software for Large-Scale Real-Time Systems," *Artificial Intelligence Review*, Vol. 25, No. 4, 2006, pp. 299–312.
doi: 10.1007/s10462-007-9028-3

[22] Sterritt, R., "Autonomic Computing," *Innovations in Systems and Software Engineering*, Vol. 1, No. 1, 2005, pp. 79–88.
doi: 10.1007/s11334-005-0001-5

[23] Nordstrom, S., Bapty, T., Neema, S., Dubey, A., and Keskinpala, T., "A Guided Explorative Approach for Autonomic Healing of Model-based Systems," *Second IEEE Conference on Space Mission Challenges for Information Technology, Mini Workshop on Autonomous and Autonomic Systems in Space Exploration*, IEEE Computer Society, Pasadena, CA, USA, July 2006.

[24] Nordstrom, S., Dubey, A., Keskinpala, T., Neema, S., and Bapty, T., "Ghost: Guided Healing and Optimization Search Technique for Healing Large-Scale Embedded Systems." *Proceedings of the Third IEEE International Workshop on Engineering of Autonomic & Autonomous Systems* (EASE'06), IEEE Computer Society, Washington, DC, USA, 2006, pp. 54–60.

[25] Russell, S. J., Norvig, P., Candy, J. F., Malik, J. M., and Edwards, D. D., *Artificial Intelligence: A Modern Approach*, Prentice-Hall, Upper Saddle River, NJ, 1996.

[26] Nilsson, N. J., *Principles of Artificial Intelligence*, Springer, Berlin, 1982.

[27] Shannon, C. E., *Programming a Computer for Playing Chess. Computer Chess Compendium*, Springer-Verlag, New York, NY, USA, 1988, pp. 2–13.

[28] Bernstein, A., and de V. Roberts, M., *Computer v Chess Player. Computer Chess Compendium*, Springer-Verlag, New York, NY, USA, 1988, pp. 43–47.

[29] Papadimitriou, C. H., "Games Against Nature," *Journal of Computer and System Sciences*, Vol. 31, No. 2, 1985, pp. 288–301.
doi: 10.1016/0022-0000(85)90045-5

[30] Gross, J. L., and Yellen, J., *Handbook of Graph Theory*, CRC Press, Boca Raton, FL, USA, 2004.

Roy Sterritt
Associate Editor