A Guided Explorative Approach for Autonomic Healing of Model-Based Systems

Steve Nordstrom, Ted Bapty, Sandeep Neema, Abhishek Dubey, Turker Keskinpala
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, USA
{steve.nordstrom, ted.bapty, sandeep.neema, abhishek.dubey, turker.keskinpala}@vanderbilt.edu

***Abstract*—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 paper reports on the progress made by our group in developing a model based toolset which specifically targets the creation of autonomic embedded systems.**

Keywords—autonomic; guided; healing; reflex-healing; model-based; model integrated; embedded;

I. INTRODUCTION

Mission critical and safety critical systems require implementations that are resilient in the face of system faults. Autonomic systems aim to provide this resiliency by adaptively mitigating potential failures. Significant design challenges arise when constructing a system capable of handling the uncertainty of multiple potential faults, occurring in arbitrary combinations and orders. Dynamically adapting a software system to meet new requirements involves far too many factors to be understood by humans in a reasonable time frame.

When a system whose design is based on models needs to be adapted to a new environment, it becomes necessary that the system have internal knowledge of its design so that it can make sense of how to alter itself. Although much work has been in the areas of rapid design of model based systems there are still situations where having a human in the redesign loop is too costly.

What follows in this paper is a discussion of a technique for allowing model-based systems to exhibit autonomic healing properties in order to solve the problems associated with large scale embedded system redesign.

II. SOFTWARE MODELING

A. Models are more than documentation

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, where advanced tools can make greater sense of compositions of model 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.

The process of modeling software allows designers to think about software using a familiar abstraction and provides a platform of understanding in a way that sharing of source code could never do. Much work has been done in the fields of computer science and electrical engineering to allow non-programmers the ability to design software by provided these familiar abstractions coupled with tools that can make sense of the designer's work and transform those design specifications into artifacts which can contribute toward an implementation.

However, in many cases, design work is 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 environments. Much of the struggle in designing these systems stems 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.

To alleviate some of these concerns, designers are using more advanced tools to design software systems. These tools use the concepts of software modeling to describe components and interactions of a system. More advanced tools can use these models for a variety of uses from validation, code synthesis, and deployment assistance.

B. Model Integrated Computing

Model Integrated Computing (MIC) [1], which has been developed at the Institute for Software Integrated Systems (ISIS) 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). 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 a bevy of useful information from the models. This information can be used for verification, simulation/analysis,

This work is supported by NSF under the ITR grant ACI-0121658.

code generation, and in other areas of a software design and deployment processes.

MIC has been shown to be an effective means of managing complexity in large scale embedded systems [2] and is being shown to allow a growing variety of analyses to be performed on models [4] [5] [6].

C. Previous work in reflex and healing architectures

The application of biologically-based two stage reflex-healing (RH) models as mechanisms for autonomicity and fault recovery in computer systems has been discussed in [7] and [8]. The application of MIC toward this problem has been discussed in [9] 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 been shown in [9] and refinements toward verification of reflexes has been shown in [10].

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 system model. If one were to examine the resulting system, much of the information regarding the relationships of the components is removed, as 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 as autonomic embedded systems), this information is of utmost importance.

Rather than attempt to design systems where this higher level knowledge is pushed down into the running system, we chose to integrate our design tools 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 in [11] to allow feedback from a live system to be included in the modeling tool. Such feedback can be used to keep the model synchronized with the system; the author recognizes the tasks of fault diagnosis and temporal model accuracy are non-trivial but beyond the scope if 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, as there are a considerable number of events which 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 un-avoidable.

In the case where the system is unable to adapt to its new environment the designer must re-visit 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 is can wait for the necessary modifications.

What happens over time is that the designers oscillate between design and re-deployment cycle after the system has been initially deployed. In the vein of adding 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 [12].

2) Retain the benefits of MIC, using the same model formats and model transformations available to a designer executing a manual redesign

3) Retain the system's ability to perform healing operations in the future

4) 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. In order 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 which is most suited to operate in the new environment. A designer has a limited set of operations she is able to perform on the model. We will consider as atomic only the operations which lead to healing. (There are many non-essential operations a designer can perform using a modeling tool; the changing of a model's color or name or other trivial operations which do not lead to healing are not considered by the automated healing tool.)

A set of allowable operations for this tools set are those which lead to healing, namely: *Promote*, *Demote*, *Transfer* (lateral transfer), *Create*, *Destroy*, *Reassociate*, *EnableTask*, and *DisableTask*. Work is ongoing to provide not just healing operations but healing strategies which can be treated as atomic operations to the Healer. This in turn will reduce the explosion of candidate healed models for a given failed model.

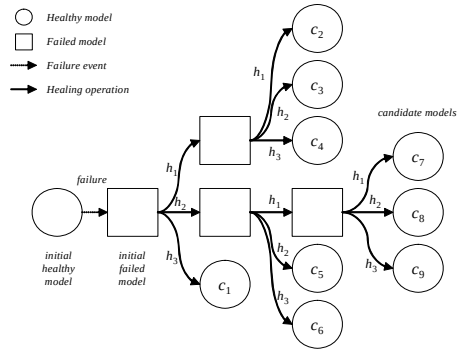

Figure 1. The determination of candidate models from an initially failed model through healing action sequences.

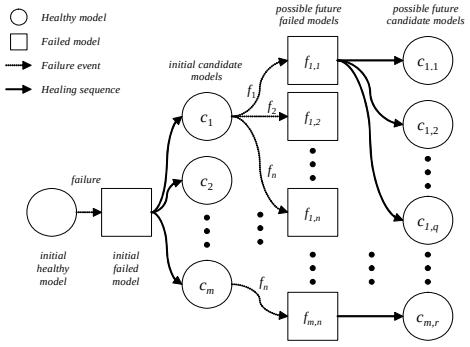

Figure 2. Determination of the most resilient model involves applying potential failures to candidate models and evaluating the resulting set of healed models.

In order to arrive at a suitable model the healer will first produce a set of candidate models in accordance with a set of healing actions which are allowed to be performed. 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 model may be considered healed, while a number may not. The procedure will continue until all the candidate models are considered healed. Fig. 1 shows this process for a failed model with three possible healing actions.

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. We have shown in [13] that this process is multi-objective in nature and that the process is dependant on the factors which drive the evaluation criteria. 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 models suitability to handle future failures.

C. The special criteria of resilience

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

The measure of resilience is made by applying all known failures to a candidate model to arrive a set of possible failure states. For each of these states the healer can perform the healing operation to form the resulting next generation of candidate models. These models are in turn evaluated for their suitability. One can see that the process is unending, so a determination must be made as to how far into the future the healer can look to determine the next healed model.

For this last stage, the healer omits the resilience criteria and evaluates with remaining models with no further failure/healing propagation. Fig. 2 shows this process in general, while Fig. 3 shows a more complete application of the look-ahead process to determine the new set of candidate models to be evaluated. Some work has been done in this area [13]; however, more clarification is needed about how a designer adapts an existing software model to accommodate change in the system.

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.

A. Domain specific modeling language

First, the Guided Healing and Optimization Search technique Modeling Language (GHOSTML) is a domain specific modeling language (DSML) used for specifying the components and interactions of the system. This is done by using three distinct aspects in which the components of the

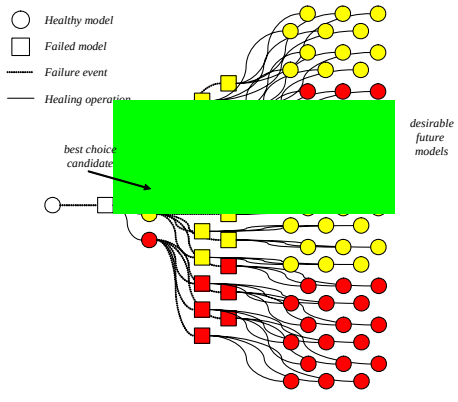

Figure 3. The best choice model is the one whose future failed and healed models are the most desirable.

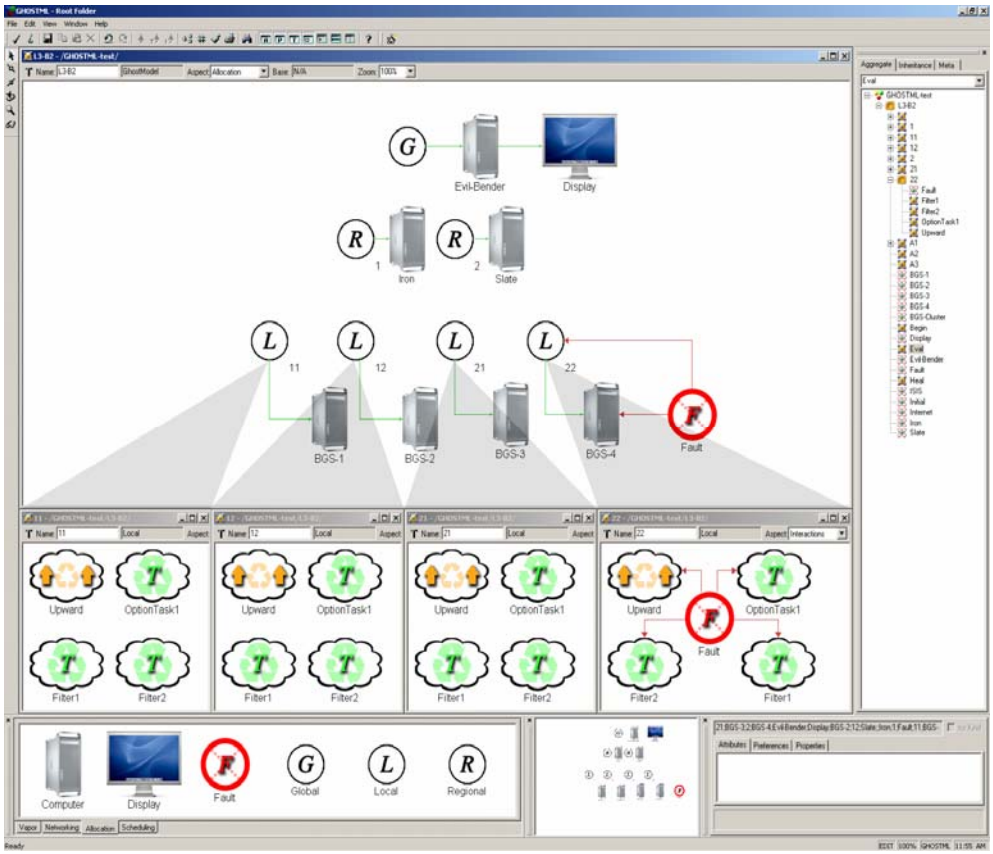

Figure 4. The modeling tools allow the capture of components in both operational and failed states. The Allocation aspect of the model is shown.

system are modeled. The aspects are the following:

- *Tasks*: A hierarchy of management tasks is created in the Tasks aspect, which determines the structure of the reflex-healing hierarchy. Containment and hierarchical decomposition of tasks are both modeling techniques which are used to manage complexity in the Tasks Aspect (e.g. managed tasks are contained inside their governing manager).
- *Networking*: Computational resources as well as data visualization and interconnect resources are modeled in the networking aspect.
- *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.

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). Fig. 5 shows a view of the Allocation aspect in GME. Since 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. Models which contain isFaulty associations and/or Fault objects are said to be *Failed* models.

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 section III which 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 section 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. Algorithms for healing and failing a models during the search are shown in Fig. 5.

The best choice candidate is the model which 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 [14] 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 [15] fashion using the multi-objective heuristic described in [13] to

```
function Heal(Model m):
   H := set of all healing operations
   max := Empty{Model, Xfm}
   Mh := {Empty{Model,Xfm}}
   plies := plies + 1
   for all h in H
      mc := ApplyXfm(h, m)
      if (plies <= maxplies)
         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 fail operations
   Mf := Empty{};
   for all f in F
      append ApplyXfm(m, f) to Mf
   return Mf
```

Figure 5. A recursive algorithm for determining resilience of candidate models is used by the Healer.

evaluate the utility of each node of the tree. Since the game is too complex to search to completion, a depth cutoff (referred to as the number of *plies*) is used to limit the scope of the look-ahead. The value of this cutoff is dependant on the move set (which therefore limits to the branching factor of the search tree) and the computational resources available to 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 move can be constructed as a composition of atomic operations but for some special reasons which will discussed later we will can consider these operation chains as a single move).

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 section III. The Fail players 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 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 [16] [17]. 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 [18] 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 as to allow a search to be conducted (as limited of depth as it may be) before the next failure occurs.

Some questions arise from this regarding the fitness of the guided search. How far into the future can the Healer look and still produce both a timely and meaningful result? Do descendants whose ancestors show high resilience to failure necessarily inherit this high resilience? These questions will soon be investigated as more statistical analyses of a running healer are performed.

V. Conclusion and Future Work

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. The strategy put forth in this paper is an attempt to integrate an entire spectrum of strategies and criteria into a single solution. 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.

ACKNOWLEDGMENT

The authors also acknowledge the contribution of other RTES collaboration team members at Fermi National Accelerator Lab, University of Illinois at Urbana-Champaign, University of Pittsburg, and Syracuse University.

REFERENCES

[1] J. Sztipanovits, G. Karsai, "Model-Integrated Computing", *IEEE Computer*, pp. 110-112, April, 1997.

[2] S. Ahuja et al., "RTES demo system2004", *ACM SIGBED Review* Special issue: The second workshop on high performance, fault adaptive, large scale embedded real-time systems (FALSE-II), vol. 2 no. 3, pp. 1-6, ISSN:1551-3688, July 2005.

[3] S. Neema, Ted Bapty, S. Shetty, S. Nordstrom, "Developing Autonomic Fault Mitigation Systems", *Journal of Engineering Applications of Artificial Intelligence Special Issue on Autonomic Computing and Grids*, Elsevier, 2004.

[4] T. Szemethy, G. Karsai, "Platform Modeling and Model Transformations for Analysis", *Journal of Universal Computing Science*, vol. 10, no. 10, pp. 1383-1408, November 23, 2004.

[5] G. Madl, S. Abdelwahed, D.C. Schmidt, "Verifying Distributed Real-time Properties of Embedded Systems via Graph Transformations and Model Checking", *Real-Time Systems*, Special Issue: Invited Papers from the 25th IEEE International Real-Time Systems Symposium (RTSS 2004), Volume 33, Numbers 1-3, Pages 77-100, July 2006.

[6] E. J. Manders, G. Biswas, N. Mahadevan, and G. Karsai, "Component-oriented Modeling of hybrid dynamic systems using the Generic Modeling environment", Proceedings of the 4th Workshop on Model-Based Development of Computer Based Systems, Potsdam, Germany, March 2006.

[7] R. Sterritt, D. Gunning, A. Meban, P. Henning, "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 (ECBS'04), pp. 449-455, 2004.

[8] R. Sterritt, D.F. Bantz, "Personal autonomic computing reflex reactions and self-healing," *Systems, Man and Cybernetics, Part C, IEEE Transactions on*, vol. 36, no. 3, pp. 304- 314, May 2006.

[9] S. Shetty, S. Nordstrom, S. Ahuja, D. Yao, T. Bapty, S. Neema, "Systems Integration of Autonomic Large Scale Systems Using Multiple Domain Specific Modeling Languages", Proceedings of the 12th IEEE International Conference on the Engineering of Computer Based Systems (ECBS 2005), 2nd IEEE Workshop on the Engineering of Autonomic Systems (EASe 2005), pp. 481-489, Greenbelt, MD , USA, April 2005.

[10] A. Dubey, S. Nordstrom, T. Keskinpala, S. Neema, T. Bapty, "Verifying Autonomic Fault Mitigation Strategies in Large Scale Real-Time Systems", Third IEEE International Workshop on Engineering of Autonomic & Autonomous Systems (EASE'06), pp. 129-140, Potsdam, Germany, March, 2006.

[11] D. Messie, M. Jung, J.C. Oh, S. Shetty, S. Nordstrom, and M. Haney, "Prototype of Fault Adaptive Embedded Software for Large-Scale Real-Time Systems", *Artificial Intelligence Review*, invited paper, special issue on Engineering Autonomic Systems, in press.

[12] R. Sterritt, "Autonomic Computing", *Innovations in Systems and Software Engineering, A NASA Journal*, vol. 1, No.1, ISSN-1614-5046, Springer, April 2005.

[13] S. Nordstrom, A. Dubey, T. Keskinpala, S. Neema, T. Bapty, "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), pp. 54-60, 2006.

[14] S. Russell, P. Norvig, *Artificial Intelligence, A Modern Approach*, Second Edition, Prentice Hall/Allyn & Bacon, 2003.

[15] N. J. Nilsson, *Principles of artificial intelligence*, San Francisco, CA, USA, Morgan Kaufmann Publishers, Inc., 1980.

[16] C. E. Shannon, "Programming a computer for playing chess", *Computer chess compendium*, Springer-Verlag New York, Inc., pp 2-13, 1988.

[17] A. Bernstein, M. de V. Roberts, "Computer v chess player", *Computer chess compendium*, Springer-Verlag New York, Inc., pp 2-13, 1988.

[18] C. H. Papadimitriou, "Games against nature", *Journal of Computer and Systems Science*, vol 31, pp. 288--301, 1985.