Automatic Recovery from Software Failure:

A Model-Based Approach to Self-Adaptive Software

Paul Robertson and Brian Williams {paulr,williams}@csail.mit.edu

MIT CSAIL, 32 Vassar Street, Building 32-272 Cambridge, MA 02139

Introduction

In complex, concurrent critical systems, every component is a potential point of failure. Typical attempts to make such systems more robust and secure are both brittle and incomplete. That is, the security is easily broken, and there are many possible failure modes that are not handled. Techniques that expand to handling component level failures are very expensive to apply, yet are still quite brittle and incomplete. This is not because engineers are lazy – the sheer size and complexity of modern information systems overwhelms the attempts of engineers, and myriad methodologies, to systematically investigate, identify, and specify a response to all possible failures of a system.

Adding dynamic intelligent fault awareness and recovery to running systems enables the identification of unanticipated failures and the construction of novel workarounds to these failures. Our approach is pervasive and incremental. It is pervasive in that it applies to all components of a large, complex system – not just the "firewall" services. It is incremental in that it coexists with existing faulty, unsafe systems, and it is possible to incrementally increase the safety and reliability of large systems. The approach aims to minimize the cost, in terms of hand-coded specifications with respect to how to isolate and recover from failures.

There are many reasons why software fails. Among the more common reasons are the following:

1. Assumptions made by the software turn out not to be true at some point. For example, if a piece of software must open a file with a given path name it will usually succeed but if the particular disk that corresponds to the path name fails the file will not be accessible. If the program assumed that the file was accessible the program will fail. In highly constrained situations it is possible to enumerate all such failures and hand code specific exception handlers – and such is the standard practice in the industry. In many cases however, particularly in embedded applications, the number of ways that the environment can change becomes so large that the programmer cannot realistically anticipate every possible failure.

2. Software is attacked by a hostile agent. This form of failure is similar to the first one except that change in the environment is done explicitly with the intent to cause the software to fail.

3. Software changes introduce incompatibilities. Most software evolves of its lifetime. When incompatibilities are inadvertently introduced software that previously did not fail for a given situation may now fail.

Whatever the reason for the software failure we would like the software to be able to recognize that it has failed and to recover from the failure. There are three steps to doing this:

1. Noticing that the software has failed;
2. Diagnosing exactly what software component has failed; and
3. Finding and alternative way of achieving the intended behavior.

In order for the runtime system to reason about its own behavior and intended behavior in this way certain extra information and algorithms must be present at runtime. In our system these extra pieces include models of the causal relationships between the software components, models of intended behavior, and models of correct (nominal) execution of the software. Additionally models of known failure modes can be very helpful but are not required. Finally the system needs to be able to sense, at least partially, its state, it needs to be able to reason about the difference between the expected state and the observed state and it need to be able to modify the running software such as by choosing alternative runtime methods.

Building software systems in this way comes with a certain cost. Models of the software components and their causal relationships that might otherwise have existed only in the programmers head must by made explicit, the reasoning engine must be linked in to the running program, and the computational cost of the monitoring, diagnosis, and recovery must be considered. In some systems the memory footprint and processor speed prohibit this approach. More and more however memory is becoming cheap enough for memory footprint to not be an issue and processor power is similarly becoming less restrictive. While the modeling effort is an extra cost there are benefits to doing the modeling that offset its cost. Making the modeling effort explicit can often cause faults to be found earlier than would otherwise be the case. The developers can choose the fidelity of the models. More detailed models take more time to develop but allow for greater fault identification, diagnosis, and recovery. Finally our approach to recovery assumes that there is more than one way of achieving a task. The developer therefore must provide a variety of ways of achieving the intended behavior.

The added costs of building robust software in this way are small when compared to the benefits.
Among the benefits it allows us to:

1. Build software systems that can operate autonomously to achieve goals in complex and changing environments;
2. Build software that detects and works around “bugs” resulting from incompatible software changes;Build software that detects and recovers from software attacks; and
4. Build software that automatically improves as better software components and models are added.

Figure 1: Deep Space 1: Fight Experiment May 1999

Prior Work: Self-adaptive software has been successfully applied to a variety of tasks ranging from robust image interpretation to automated controller synthesis [7]. Our approach, which is

described below, builds upon a successful history of hardware diagnosis and repair. In May 1999 the spacecraft Deep Space 1 [1] ran autonomously for a period of a week. During that week faults were introduced which were detected, diagnosed, and recovered from by reconfiguring the (redundant) hardware of the spacecraft. Subsequently Earth Observer 1 has been flying autonomously planning and executing its own missions. Extending these technologies to software systems involves extending the modeling language to deal with the idiosyncrasies of software such as its inherently hierarchical structure [6].

Approach: At the heart of our system is a model-based programming language called RMPL that provides a language for specifying correct and faulty behavior of the systems software components. The novel ideas in our approach include **method deprecation** and **method regeneration** in tandem with an intelligent runtime model-based executive that performs **automated fault management** from engineering models, and that utilizes decision-theoretic method dispatch. Once a system has been enhanced by abstract models of the nominal and faulty behavior of its components, the model-based executive monitors the state of the individual components according to the models. If faults in a system render some methods (procedures for accomplishing individual goals) inapplicable, method deprecation removes the methods from consideration by the decision-theoretic dispatch. Method regeneration involves repairing or reconfiguring the underlying services that are causing some method to be inapplicable. This regeneration is achieved by reasoning about the consequences of actions using the component models, and by exploiting functional redundancies in the specified methods. In addition, decision-theoretic dispatch continually monitors method performance and dynamically selects the applicable method that accomplishes the intended goals with maximum safety, timeliness, and accuracy.

Beyond simply modeling existing software and hardware components, we allow the specification of high-level methods. A method defines the intended state evolution of a system in terms of goals and fundamental control constructs, such as iteration, parallelism, and conditionals. Over time, the more that a system's behavior is specified in terms of model-based methods, the more that the system will be able to take full advantage of the benefits of model-based programming and the runtime model-based executive. Implementing functionality in terms of methods enables method prognosis, which involves proactive method deprecation and regeneration, by looking ahead in time through a temporal plan for future method invocations.

Our approach has the benefit that every additional modeling task performed on an existing system makes the system more robust, resulting in substantial improvements over time. As many faults and intrusions have negative impact on system performance, our approach also improves the performance of systems under stress.

Our approach provides a well-grounded technology for incrementally increasing the robustness of complex, concurrent, critical applications. When applied pervasively, model-based execution will dramatically increase the security and reliability of these systems, as well as improve overall performance, especially when the system is under stress.

Fault Aware Processes Through Model-based Programming

Recall, to achieve robustness pervasively, fault adaptive processes must be created with minimal programming overhead. *Model-based programming* elevates this task to the specification of the intended state evolutions of each process. A *model-based executive* automatically synthesizes fault adaptive processes for achieving these state evolutions, by reasoning from models of correct and faulty behavior of supporting components.

Each model-based program implements a system that provides some service, such as secure data transmission. This is used as a component within a larger system. The model-based program in turn builds upon a set of services, such as name space servers and data repositories, implemented through a set of concurrently operating components, comprised of software and hardware.

Component Services Model

The *service model* represents the normal behavior and the known and unknown aberrant behaviors of the program's component services. It is used by a deductive controller to map sensed variables to queried states. The service model is specified as a concurrent transition system, composed of probabilistic concurrent constraint automata 1. Each component automaton is represented by a set of component modes, a set of constraints defining the behavior within each mode, and a set of probabilistic transitions between modes. Constraints are used to represent co-temporal interactions between state variables and intercommunication between components. Constraints on continuous variables operate on qualitative abstractions of the variables, comprised of the variable's sign (positive, negative, zero) and deviation from nominal value (high, nominal, low). Probabilistic transitions are used to model the stochastic behavior of components, such as failure and intermittency. Reward is used to assess the costs and benefits associated with particular component modes. The component automata operate concurrently and synchronously.

Self Deprecation and Regeneration Through Predictive Method Dispatch

In model-based programming, the execution of a method will fail if one of the service components it relies upon irreparably fails. This in turn can cause the failure of any method that relies upon it, potentially cascading to a catastrophic and irrecoverable system-wide malfunction. The control sequencer enhances robustness by continuously searching for and deprecating any requisite method whose successful execution relies upon a component that is deemed faulty by mode estimation, and deemed irreparable by mode reconfiguration.

Without additional action, a deprecated method will cause the deprecation of any method that relies upon it, potentially cascading to catastrophic system-level malfunction. Model-based programmers specify redundant methods for achieving each desired function. When a requisite method is deprecated, the control sequencer attempts to regenerate the lost function proactively, by selecting an applicable alternative method, while verifying overall safety of execution.

More specifically, *predictive method selection* will first search until it finds a set of methods that are consistent and schedulable. It then invokes the dispatcher, which passes each activity to the deductive controller as configuration goals, according to a schedule consistent with the timing constraints. If the deductive controller indicates failure in the activity's execution, or the dispatcher detects that an activity's duration bound is violated, then method selection is reinvoked. The control sequencer then updates its knowledge of any new constraints and selects an alternative set of methods that safely completes the RMPL program.

Self-Optimizing Methods Through Safe, Decision-Theoretic Dispatch

In addition to failure, component performance can degrade dramatically, reducing system performance to unacceptable levels. To maintain optimal performance, predictive method dispatch utilizes decision-theoretic method dispatch, which continuously monitors performance, and selects the currently optimal available set of methods that achieve each requisite function.

Results

Initial testing of the described system has been performed by augmenting the MIT MERS rover test bed. The rover test bed consists of a fleet of ATRV robots within a simulated Martian terrain. By way of example we describe one mission whose robustness has been enhanced by the system.

Two rovers must cooperatively search for science targets in the simulated Martian terrain. This is done by having the rovers go to the selected vantage points looking for science targets using the rover's stereo cameras. The rovers divide up the space so that they can minimize the time taken in mapping the available science targets in the area. The paths of the rovers are planned in advance given existing terrain maps. The plan runs without fail. Between them the rovers successfully find all of the science targets that we have placed for them to find. The scenario is shown below in Figure 2.

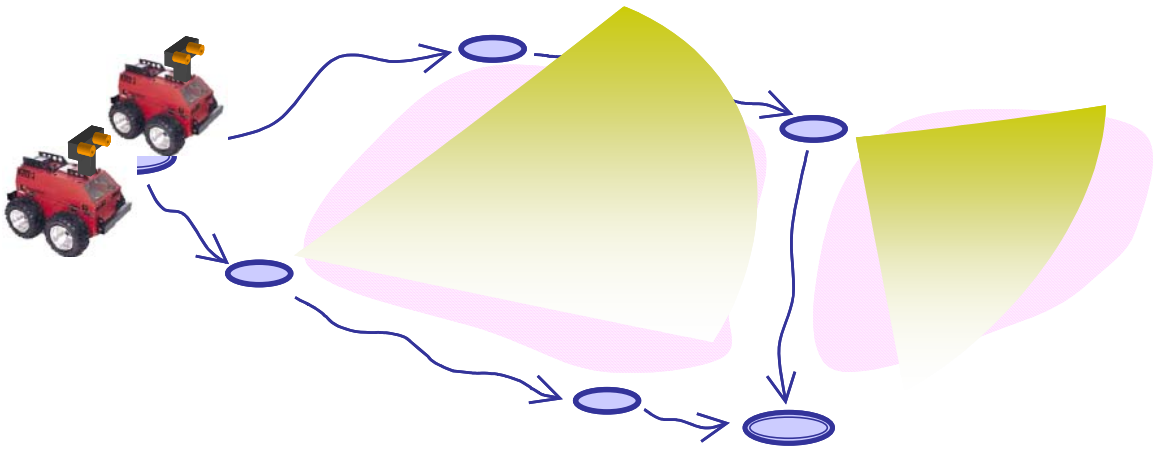

Figure 2: Rover test bed experimental platform

In the test scenario two faults are introduced by placing a large rock that blocks rover #1's view of one of the designated areas. When rover #1 gets into its initial position to look for science targets its stereo cameras detect the unexpected rock obscuring its view. This results in an exception that disqualifies the current software component from looking for targets. Since the failure is external to the rover software the plan itself is invalidated. The exception is resolved by replanning which allows the both rovers to modify their plans so that the second rover observes the obscured site from a different vantage point. The rovers continue with the new plan but when rover #2 attempts to scan the area for science targets the selected vision algorithm fails due to the deep shadow being cast by the large rock. Again an exception is generated but in this case a redundant method is found – a vision algorithm that works well in low light conditions. With this algorithm the rover successfully scans the site for science targets. Both rovers continue to execute their plan without further failure.

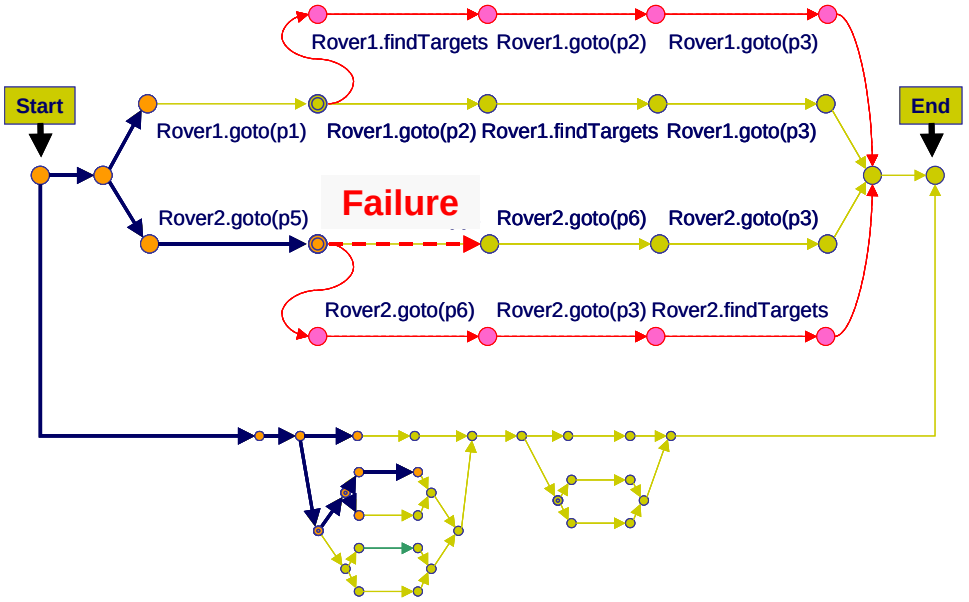

Figure 3: The TPN for the two-rover exploration plan. Failure due to an obscuration (rock) results in automatic online replanning so that the mission can continue.

Comparison with Current Technology

Model-based Programming of Hidden States

The reactive model-based programming language (RMPL) is similar to reactive embedded synchronous programming languages like Esterel. In particular, both languages support conditional execution, concurrency, preemption and parameter less recursion. The key difference is that in embedded synchronous languages, programs only read sensed variables and write to controlled variables. In contrast, RMPL specifies goals by allowing the programmer to read or write ``hidden" state variables. It is then the responsibility of the language's model-based execution kernel to map between hidden states and the underlying system's sensors and control variables.

Predictive and Decision-theoretic Dispatch

RMPL supports nondeterministic or decision theoretic choice, plus flexible timing constraints. Robotic execution languages, such as RAPS, [2], ESL[4] and TDL[3], offer a form of decision theoretic choice between methods and timing constraints. In RAPS, for example, each method is assigned a priority. A method is then dispatched, which satisfies a set of applicability constraints while maximizing priority. In contrast, RMPL dispatches on a cost that is associated with a dynamically changing performance measure. In RAPS timing is specified as fixed, numerical values. In contrast, RMPL specifies timing in terms of upper and lower bound on valid execution times. The set of timing constraints of an RMPL program constitutes a Simple Temporal Network (STN). RMPL execution is unique in that it predictively selects a set of future methods whose execution are temporally feasible.

Probabilistic Concurrent Constraint Automata

Probabilistic Concurrent Constraint Automata (PCCA) extend Hidden Markov Models (HMMs) by introducing four essential attributes. First, the HMM is factored into a set of concurrently operating automata. Second, probabilistic transitions are treated as conditionally independent. Third, each state is labeled with a logical constraint that holds whenever the automaton marks that state. This allows an

efficient encoding of co-temporal processes, which interrelate states and map states to observables. Finally, a reward function is associated with each automaton, and is treated as additive.

Constraint-based Trellis Diagram

Mode estimation encodes PHCA as a constraint-based trellis diagram, and searches this diagram in order to estimate the most likely system diagnoses. This encoding is similar in spirit to a SatPlan/Graphplan encoding in planning.

Conclusions

We have extended a system capable of diagnosing and reconfiguring redundant hardware systems so that instrumented software systems can likewise be made robust. Software systems are more complex than hardware systems and modeling software components and their interconnections poses a higher modeling burden. Results of our early experiments are encouraging but much work remains to extend the current experimental system to cover the full range of software practice.

Acknowledgements

The work described in this article was supported in part by DARPA and NASA.

References

1. D. Bernard, G. Dorais, E. Gamble, B. Kanefsky, J. Kurien, G. Man, W. Millar, N. Muscettola, P. Nayak, K. Rajan, N. Rouquette, B. Smith, W. Taylor, Y. Tung, Spacecraft Autonomy Flight Experience: The DS1 Remote Agent Experiment, Proceedings of the AIAA Space Technology Conference & Exposition, Albuquerque, NM, Sept. 28-30, 1999. AIAA-99-4512.
2. R. Firby, "The RAP language manual," Working Note AAP-6. University of Chicago, 1995.
3. R. Simmons, "Structured Control for Autonomous Robots," *IEEE Transactions on Robotics and Automation*, 10(1), 94.
4. E Gat, "ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents," In Proceedings of the AAAI Fall Symposium on Plan Execution, 1996.
5. B. Williams and P. Nayak. A Reactive Planner for a Model-based Execution. In Proceedings 15th International Joint Conference AI, Nagoya, Japan, August 1997. IJCAI-97.
6. T. Mikaelian & M. Sachenbacher. Diagnosing Mixed Hardware/Software Systems Using Constraint Optimization. In Proceedings DX-05.
7. Robert Laddaga, Paul Robertson, Howard E. Shrobe. Introduction to Self-adaptive Software: Applications. Springer Verlag LNCS 261