AUTOMATIC RECOVERY FROM SOFTWARE FAILURE

By PAUL ROBERTSON *and* BRIAN WILLIAMS

A model-based approach to self-adaptive software.

In complex concurrent critical systems, such as autonomous robots, unmanned air vehicles, and space systems, every component is a potential point of failure. This is true not only of embedded systems but also of purely software systems such as distributed and cyber applications. Typical attempts to make such systems more robust and secure are both brittle and incomplete due to reliance on manual identification of and solutions to potential failures such as by using exception mechanisms. That is, the security is easily broken, and there are many possible failure modes that are not handled. Failures may be rare events so it is less easy to test for good coverage of fault scenarios. 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.

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.

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, the most common include:

- 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 assumes that the file is 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 the environment can change becomes so large that the programmer cannot realistically anticipate every possible failure.
- 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.
- Software changes introduce incompatibilities. Most software evolves during 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: noticing the software has failed; diagnosing exactly what software component has failed; and finding an alternative way of achieving the intended behavior.

Approach

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 must be able to sense, at least partially, its state; to reason about the difference between the expected state and the observed state; and to modify the running software (for example, 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, which might otherwise have existed only in the programmer's head, must be made explicit; the reasoning engine must also 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. However, memory is increasingly becoming cheap enough for memory footprint not to be an issue; processor power is similarly becoming less restrictive. While the modeling effort adds 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, and 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 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 build software that:

- Operates autonomously to achieve goals in complex and changing environments;
- Detects and works around "bugs" resulting from incompatible software changes;
- Detects and recovers from software attacks; and
- Automatically improves as better software components and models are added.

At the heart of our system is Reactive Model-based Programming Language (RMPL), a language for specifying correct and faulty behavior of the system's software components. The novel ideas in our approach include the use of *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 inapplicable, method deprecation removes them 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 (iteration, parallelism, and conditionals). Over time, the more a system's behavior is specified in terms of model-based methods, the more it 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. It provides a well-grounded technology for incrementally increasing the robustness of complex, concurrent, critical applications. When applied pervasively, model-based execution can 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

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) 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 that consist of software and hardware.

Component Service Model. The *service model* represents the normal behavior and the known aberrant behaviors of the program's component services. Unknown aberrant behaviors are also supported by the service model through the inclusion of unmodeled failure modes. 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. 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, characterized by 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.

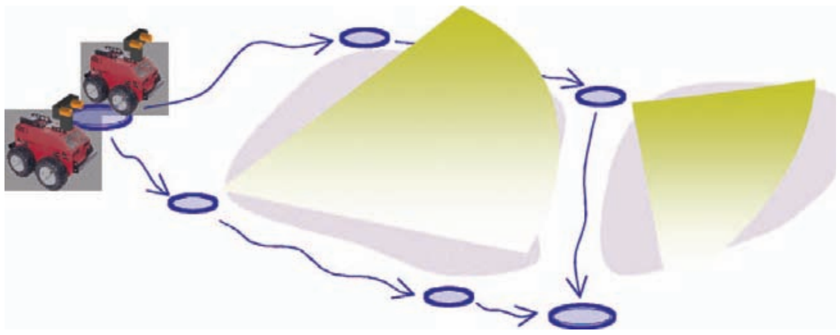


Figure 1. Rover testbed experimental platform.

Self-Deprecation and Regeneration through Predictive Method Dispatch. In model-based programming, the execution of a method fails 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 systemwide 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 causes the deprecation of any method that relies upon it. 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* first searches 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, the method selection component is re-invoked. The control sequencer then updates its knowledge of any new constraints and selects an alternative set of methods that safely complete 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 Model-Based Embedded and Robotic Systems rover testbed. The rover testbed consists of a fleet of all-terrain robot vehicles 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 targets of interest using the rover's stereoscopic cameras. The rovers divide up the space so 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 in Figure 1.

In the test scenario, two faults are introduced by placing a large rock that blocks Rover1's view of one of the designated areas. When Rover1 reaches its initial position to look for science targets, its cameras detect the unexpected rock obscuring its view. This

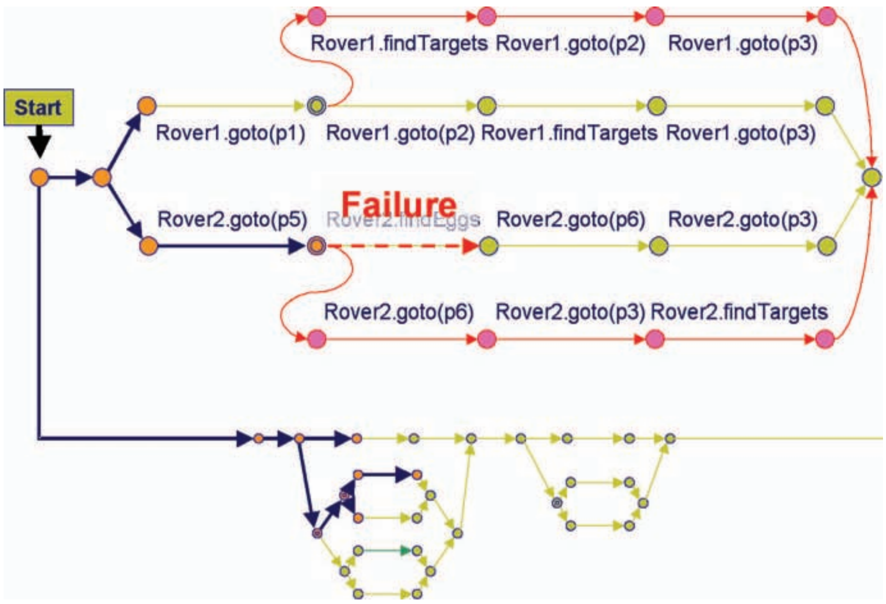
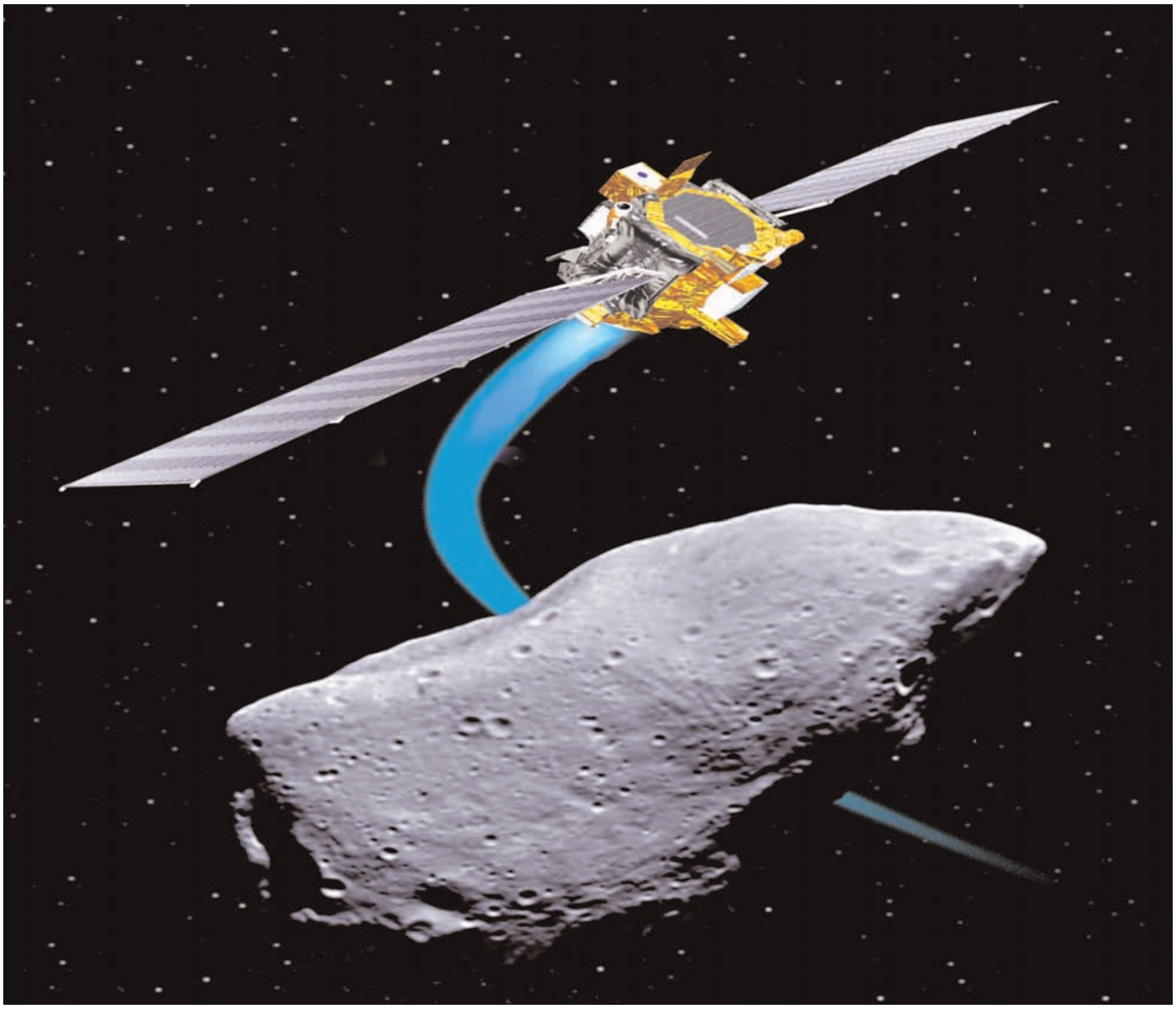


Figure 2. (left) The temporal planning network for the two-rover exploration plan. Failure due to an obscuration (rock) results in automatic replanning so that the mission can continue.

Figure 3. (below) Deep Space 1: Fight experiment (May 1999).



SELF-ADAPTIVE SOFTWARE HAS BEEN SUCCESSFULLY APPLIED TO A VARIETY OF TASKS, RANGING FROM ROBUST IMAGE INTERPRETATION TO AUTOMATED CONTROLLER SYNTHESIS.

results in an exception that disqualifies the current software component from looking for targets. Because the failure is external to the rover software, the plan itself is invalidated. The exception is resolved by replanning, which allows both rovers to modify their plans so that Rover2 observes the obscured site from a different vantage point. The rovers continue with the new plan but when Rover2 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 the revised plan without further failure (see Figure 2).

Related Work

Self-adaptive software has been successfully applied to a variety of tasks, ranging from robust image interpretation to automated controller synthesis [4]. Our approach builds on a successful history of hardware diagnosis and repair [7]. In May 1999, the spacecraft Deep Space 1, shown in Figure 3, ran autonomously for a period of one week [1]. During that week, faults were introduced that were detected, diagnosed, and resolved by reconfiguring the (redundant) hardware of the spacecraft. Subsequently, another satellite (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 [5].

Model-based Programming of Hidden States. RMPL is similar to reactive embedded synchronous programming languages such as 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 [3], and TDL [6] 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 is 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 interrelates states and maps states to observables. Finally, a reward function is associated with each automaton.

Constraint-based Trellis Diagram. Mode estimation encodes Probabilistic Hierarchical Constraint Automata (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.

Conclusion

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 lack many of the attributes of hardware systems to which the described methods have traditionally been applied; they tend to be more hierarchical and have more complex and numerous component interactions. Software components and their interconnections represent a significantly higher modeling burden.

Our approach differs from other similar techniques in the following ways:

- Models specify program behavior in terms of abstract states, which simultaneously makes the models easier to read and think about and somewhat robust to changes in low-level software implementation decisions.
- Modeling covers a wide spectrum of software considerations from a high-level storyboarding of the software to temporal considerations, if any, to the causal relationships between components.
- Robustness and recovery derives from a collection of complex and highly tuned reasoning algorithms that estimate state, choose contingencies, and plan state trajectories. The programmer is largely shielded from this complexity because the mechanism is hidden behind the intuitive unified modeling language.

An interesting feature of our approach is the ability to add robustness incrementally. More modeling leads to greater runtime robustness because it allows the system to detect, diagnose, and repair more fault situations. This means the effort devoted to modeling can be managed in much the same way that is done for test suite development in conventional software development projects.

Modeling errors can result in a number of undesirable outcomes such as failure to detect fault conditions and subsequent failure to recover from the fault, incorrect diagnosis of the fault, and attributing faults to components that are operating correctly. In this sense an incorrect model is no different from any other bug in the software. It is somewhat easier to deal with, however, because the models are written at a more abstract level than the program itself, making them easier to read. There is a problem with making changes to the software definition and neglecting to update the models of the software. In time we expect tools to evolve to address this kind of problem.

The nature of the models developed for a software system vary depending upon the nature of the software itself. Some programs, especially those involved in embedded and robotic applications, have critical timing considerations that must be modeled as such whereas other programs have no timing of synchronization considerations.

Developing model-based reconfigurable software systems is a relatively new endeavor, but results of our early experiments are encouraging. Much work remains to extend the current experimental system to cover the full range of software practice. C

References

1. Bernard, D., Dorais, G., Gamble, E., et al. Spacecraft autonomy flight experience: The DS1 remote agent experiment. In *Proceedings of the AIAA Space Technology Conference and Exposition* (Albuquerque, NM, Sept. 1999).
2. Firby, R. *The RAP Language Manual*. Working Note AAP-6, University of Chicago, 1995.
3. Gat, E. ESL: A language for supporting robust plan execution in embedded autonomous agents. In *Proceedings of the AAAI Fall Symposium on Plan Execution* (Cambridge, MA, Nov. 1996).
4. Laddaga, R., Robertson, P., and Shrobe, H.E. Introduction to self-adaptive software: Applications. In *Proceedings of the 2nd International Workshop on Self-Adaptive Software (IWSAS 2001)*, (Balatonfüred, Hungary, May 2001), LNCS 2614, Springer.
5. Mikaelian, T., Williams, B.C., and Sachenbacher, M. Diagnosing complex systems with software-extended behavior using constraint optimization. In *Proceedings of the 16th International Workshop on Principles of Diagnosis (DX-05)*, (Monterey, CA, June 2005).
6. Simmons, R. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation 10*, 1 (1994), 34–43.
7. Williams, B. and Nayak, P. A reactive planner for a model-based execution. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, (Nagoya, Japan, August 1997).

Paul Robertson (paulr@csail.mit.edu) is a research scientist at the Massachusetts Institute of Technology's Computer Science and Artificial Intelligence Laboratory, Cambridge, MA.
Brian Williams (williams@mit.edu) is Boeing Associate Professor of Aeronautics and Astronautics at the Massachusetts Institute of Technology, Cambridge, MA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

© 2006 ACM 0001-0782/06/0300 $5.00