A Model-Based System
Supporting Automatic
Self-Regeneration of Critical
Software
Paul Robertson & Brian Williams
Model-Based and Embedded Robotic Systems
http://mers.mit.edu
MIT
Computer Science and Artificial Intelligence Laboratory

What we are trying to do
- Why software fails:
  - Software assumptions about the environment become invalid because of changes in the environment.
  - Software is attacked by a hostile agent.
  - Software changes introduce incompatibilities.
- What can be done when software fails:
  - Recognize that a failure has occurred.
  - Diagnose what has failed – and why.
  - Find an alternative way of achieving the intended behavior.

Self repairing explorer:
Deep Space 1

Project Status
Funding: DARPA (SRS), NASA (Ames)
Current State: Prototype System Operational
Project Premise:
  Extend proven approach to hardware diagnosis and repair as used in DS-1 to critical software.
Principle Ideas:
  Model-Based Language Approach
  Redundant Methods
  Method Deprecation
  Model-Predictive Dispatch
  Hierarchical Models
  Adjustable Autonomy

Overview
Technical Objective:
When software fails because (a) environment changes (b) software incompatibility (c) hostile attack, (1) recognize that a failure has occurred, (2) diagnose what has failed and why, and (3) find an alternative way of achieving the intended behavior.
Technical approach:
By extending RMPL to support software failure, we can extend robustness in the face of hardware failures to robustness in the face of software failures. This involves:
  (1) Detection
  (2) Diagnosis
  (3) Reconfiguration
  (4) Utility Maximization

RMPL Models of:
Software Components, Component Hierarchy & Interconnectivity, and Correct Behavior.
Expected Benefits

- Software systems that can operate autonomously to achieve goals in complex and changing environments.
  - Modeling environment
- Software that detects and works around “bugs” resulting from incompatible software changes.
  - Modeling software components
- Software that detects and recovers from software attacks.
  - Modeling attack scenarios
- Software that automatically improves as better software components and models are added.

What can go wrong?

2. Software: A problem with the environment.
   1. A mismatch between a chosen algorithm and the environment such as there not being enough light to support processing of a color image.
   2. An unexpected imaging problem such as an obstruction to the visual field (caused by a large obscuring rock).

Solution to 2.1
- Reconfigure the software structure:
  1. Redundant Methods
  2. Mode Estimation
  3. Mode Reconfiguration

Solution to 2.2
- Switch to a contingent plan:
  1. Exception
  2. Model Predictive Dispatch
  3. Replanning

Test Bed Platform

Involves:
- Cooperative use of multiple robots.
- Timing critical software.
- Reconfiguration of Software Components.
  - Multiple Redundant Methods
  - Continuous Replanning

Science Target Search Scenario

- Cooperatively search for targets in the predefined regions
- Search from predefined viewpoints
- Search for the targets using stereo cameras and various visualization algorithms

Science Target Search Scenario
Method Regeneration: Exception Handling

- A rock blocks the view
  - Recover by taking the image from a different perspective (i.e. change the strategy)
- The shadow cast by the rock fails the imaging code from identifying the objects in view
  - Reconfigure the imaging algorithm to work under these conditions
Overall Architecture

Reconfigurable Vision for Robust Rover Mapping

Reconfigurable Vision Plant Model

Nominal Configuration

Contingent Configuration

Connection

Models

Plan Runner

Deductive Controller

Mode Estimation

Mode Reconfiguration

Plant

Inputs: x

Outputs: x

class Connection (){
    RawImage image_in;
    SegmentedImage image_out;
    mode Connected (...) { primitive method disconnect (...) => Unconnected; } mode Unconnected (...) { primitive method connect (...) => Connected; failure mode Failed (...) { ... }; }
}
class SegmentColor ()
{
    RawImage image_in;
    SegmentedImage image_out;
    mode Usable ((image_in = Nominal)) { ... }
    mode TooDark ((image_in = Dark)) { ... }
}

Solution Analysis: Exception Handling
1. Execution begins...
2. An error occurs, and an exception is thrown
3. The exception-handling code is inserted
4. Replanning begins, pre-selecting anything that has already been executed

Conclusions
- Models of correct operation permits:
  - Detection and Diagnosis of failed components.
  - Reconfiguration of Software/Hardware components to achieve high-level goals
  - Describe goals as abstract state trajectories.
- Software can be handled by adding:
  - Hierarchy to component organization
  - Models of the environment