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