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