Specification-Driven Prototyping for Architecting Dependability

Dennis B. Mulcare, ACM member
Consultant
cefsm@ellijay.com

Abstract - This paper describes a major part of an architecting methodology developed for safety-critical fault-tolerant software systems. The methodology coverage centers on specification-driven prototyping. This approach to prototyping is seen to be superior to the customary approaches of throwaway and evolutionary prototyping. A still developmental form of representation, higher-level statecharts, provides a suitably expressive prototype specification language.

Dependability is held to rely crucially on the rigor and specificity of the architecting process, as well as on the propagatability of its products. The subject four-step prototyping approach can subserve such needs, especially with regard to conceptualization insights, complexity management, dynamic analysis, and dependability assurances. Such efforts primarily address the underlying architecture or infrastructure of a nascent software system. In particular, the advocated prototyping approach focuses on absolute time-based concurrency, with accommodation of arbitrary scalability, non-ideal timing, and stochastic effects.

1 INTRODUCTION

Dependability refers to an encompassing qualitative judgment regarding the degree to which a software system merits or elicits the confidence of customers and users. Mainly, dependability resides in the extra-functional properties exhibited by a deployed system, and ultimately, in the system's underlying architecture or infrastructure. The underlying architecture includes most of the redundancy elements for fault tolerance, along with system-wide management logic. The definition and organization of such features during architecting present major leverage for ensuring high dependability from the outset of development.

Driven primarily by the demands of safety-critical systems, the development of associated methods has proceeded, with moderate success, for some three decades now. The routine use of these methods, however, has largely been restricted to embedded applications that prompt their use because of the severity of inherent hazards. Nonetheless, the adequacy of dependability methods and practices remains a continuing challenge because of the ever-increasing degree of sophistication and integration sought in embedded software systems.

1.1 Critical System Background

A dependable development process with reliable methods is essential to the assured development of dependable software systems. Safety-critical embedded systems, such as fly-by-wire flight control systems, motivated extensive research in associated development methods from the 1970s well into the 1980s. More or less independently, software-based prototyping methods evolved during the same period.

As evident by deployed aerospace systems, critical system development practice has been rather successful from about 1980 forward. Such success has in general been achieved because the associated software systems were:

- Dedicated to the critical function(s);
- Kept as simple as possible;
- Constrained in by limited computing resources;
- Supported by specialized hardware;
- Developed by staff with *a priori* understanding of the intended system needs and capabilities;
- Driven by considerations of safety and reliability.

In short, dependability was explicitly a major driver in architecting, implementing, and deploying such systems. While safety criticality was the paramount concern, other properties engendering dependability were necessarily and consciously incorporated into such systems. For example, various self-test and fault diagnosis features essential to safety also enhanced reliability and maintainability. Also, the criticality of real-time constraints levied stringent demands on infrastructure performance.

1.2 Current Dependability Challenges

Then, as processing power and functional integration became compelling realities, new issues regarding dependability arose because typically the physical segregation of critical functions was no longer programmatically viable. In turn, the requirements placed on the infrastructure became increasingly more diverse, demanding, and complex.

Regarding both research and practice, probably the greatest opportunity for dependability technology improvement exists at the architecting stage. Apart from the more customary concerns over functionality or applications, assured dependability mainly necessitates a rigorous approach to the formulation, analytical assessment, and verification of a nascent software system's underlying architecture. Typically, safety-critical systems involve redundancy management and hard real-time constraints that compel a focus on the dynamic analysis of concurrency logic and absolute timing in order to ensure requisite levels of dependability.

Such architecting activities can be facilitated through specification-driven prototyping, as depicted in Figure 1. Here, all prototype definition and development is first performed at the specification level. Then, the prototype is implemented or modified accordingly. Its execution is used in problem exploration, architecture development, dynamic analysis, and concept validation. Ultimately, the verified and optimized results, especially quantitative parameters as for timing, are propagated from the prototype specification to that of development product.

For the identified class of problems, however, two major problems persist regarding readily usable prototyping methods. First, there is the problem of a specification language that can express absolute time-based concurrency in an arbitrarily scalable manner. Such scalability should apply to both process types and data types needed in prototype specification. Second, specifications rendered in such a language should be directly and precisely translatable into an executable form. Such capacity is vital to methods automation. These issues, together with the inclusion of stochastic effects, appear to be neglected areas of research. These issues, moreover, would seem to be relevant to many classes of systems with significant dependability demands.

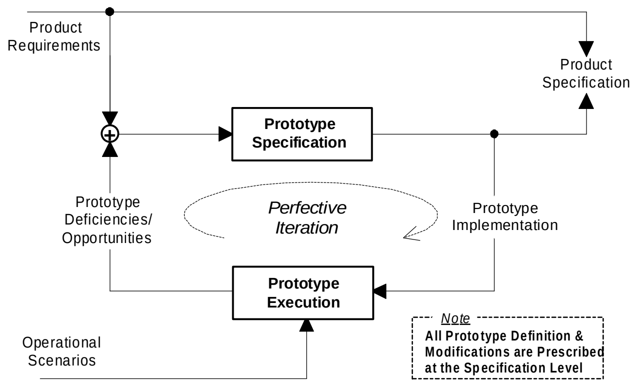


Figure 1. Specification-Driven Prototyping Concept

2 ARCHITECTING METHODOLOGY

A dependable architecting process, supported by reliable and effective methods, is necessary to ensure a dependable development product. To stabilize subsequent development and manage complexity throughout system development, the architecting process should capture all the *essential* problem complexity at the outset. Where demands on the architecture are stringent, the methodology should also be suitably rigorous and tailored to the particular system characteristics. For safety-critical embedded software systems then, the architecting methodology should preferably concentrate on system-wide control logic and address the underlying architecture largely separate from that of the applications architecture. This separation is generally quite tractable, and is beneficial for both static and dynamic analysis.

Table 1 summarizes salient attributes of the applications versus the infrastructure architectures, which together compose the software system architecture. Dependability associates mainly with the extra-functional requirements, which for safety-critical systems tend to have Go/No-Go acceptability criteria. Thus, the proper operation of the infrastructure must be sustained under both faulted and fault-free conditions, generally under very stringent absolute timing constraints. Moreover, the operation of the applications architecture is totally dependent on that of the infrastructure. Further, the acceptability criteria for the operation of the applications architecture are usually relatively tolerant, with some latitude for limited functional performance degradation.

From an architecting standpoint, the distinction in Table 1 between System STATE and Operational MODE is pivotal. System state denotes the state of the overall system, and hence the support available for functionality. On the other hand, operational mode effects the activation of particular functionality, conditional upon its availability per system state. System state is based on absolute time based concurrency logic that manages the system and its hardware elements. As such, the real-time determination of system is state is both critical and intricate, especially where diverse hardware operability and fault tolerance are involved.

As a consequence, the overall methodology that is context for this paper is called control state decomposition[1]. Practitioners in safety-critical systems have employed this kind of methodology, at least implicitly, to some degree. Otherwise, the linkage from dependability-related analyses to the development product is tenuous or obscure, and hence potentially misleading. Control state decomposition is relevant here for several reasons:

- Infrastructure-applications partitioning;
- Dependability-related product specification content;
- Focus of dynamic modeling and analysis.

APPLICATIONS ARCHITECTURE	INFRASTRUCTURE ARCHITECTURE
Functional Requirements	Extra-functional Requirements
System Services	System Properties ("Ilities")
What Kind(s) of Service	How Well Service is Supported
Operational MODE	System STATE
Functional Performance	Infrastructure Performance
Shades of Grey Criteria	GO/NO-GO Criteria

Table 1. Applications versus Infrastructure Architecture

2.1 Applications-Infrastructure Dichotomy

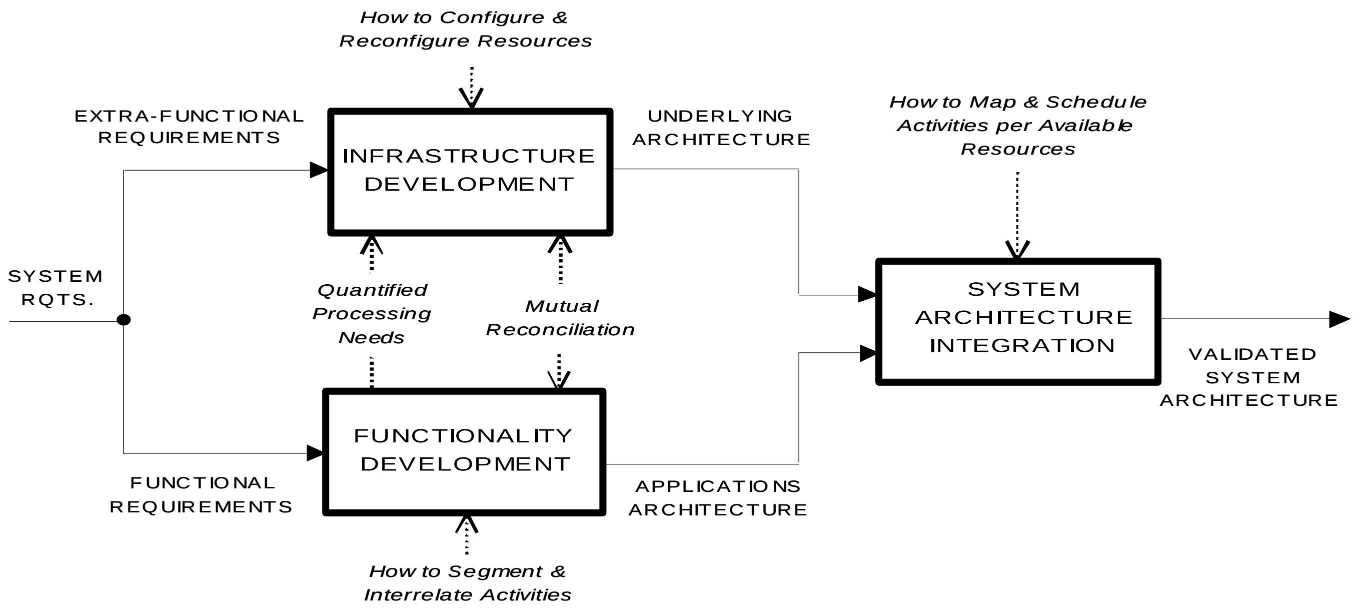
As seen in Figure 2, the architecting accomplished under the control state decomposition methodology begins with a delineation of software system requirements into functional and extra-functional ones. On separate but interdependent trajectories, the functional requirements drive the architecting of the applications architecture, while the extra-functional requirements drive the development of the underlying architecture. It serves as the system-wide platform used by the applications architecture.

The extra-functional requirements are typically rather high-level statements, usually quantitative ones. Hence, these requirements are subject to considerable interpretation and elaboration. Their interpretation largely dictates the degree of rigor and kinds assurance methods that need to be used in development. Their elaboration derives from the analysis and architecting associated with the functional requirements. As seen in Figure 2, the applications architecture furnishes its quantitative processing needs for infrastructure development. These needs are identified for all operational modes, with emphasis on worst-case demands.

The structure *per se* of the underlying architecture, such as redundancy management schemes and self-test mechanisms, derive largely from the extra-functional requirements. The suitability of that structure, however, is subject to the requirements derived through applications architecting. Further, the parameterization of that structure, such as capacity or timing quantities, is determined by the needs of the applications architecture. As suggested earlier, the interplay between the underlying and the applications architectures is mediated by the system state and operational mode logic. Associated information flow involves sensor, controller, display, and effector signals as essential to the intended system functionalities.

The system architecture integration shown in Figure 2 is a logical one that centers on dynamic analysis of various applications mode demands. The focus is on overall quantitative adequacy under worst-case demands. Cases of particular concern are the transient workload for handling faults and potential computational delays due to dependencies in the interleaving of distributed processes. At this stage, the associated software exists only in logical or specification forms, so the associated dynamic analysis is truly at an architectural level.

Such analysis involves prototype execution, or absolute time-based simulation, using representative and worst-case scenarios. Simulation provides tangible insights and quantitative calibrations of architectural acceptability, including information on behaviors pertaining to dependability. Such information can be vital to associated static analysis in terms of data or assumptions.



How to Configure & Reconfigure Resources
EXTRA-FUNCTIONAL REQUIREMENTS
INFRASTRUCTURE DEVELOPMENT
UNDERLYING ARCHITECTURE
How to Map & Schedule Activities per Available Resources
SYSTEM RQTS.
Quantified Processing Needs
Mutual Reconciliation
SYSTEM ARCHITECTURE INTEGRATION
VALIDATED SYSTEM ARCHITECTURE
FUNCTIONALITY DEVELOPMENT
FUNCTIONAL REQUIREMENTS
APPLICATIONS ARCHITECTURE
How to Segment & Interrelate Activities

Figure 2. Architecting Methodology Dichotomy

2.2 The Architecting of Dependability

During software system architecting, dependability is pursued indirectly through ensuring the aforementioned extra-functional requirements. For example, the fulfillment of safety, and reliability may be explicitly sought in composing a software system architecture. Pending architecture commitments then have to be analyzed to predict or confirm compliance with the respective requirements. All such assessments are not quantitative, as safety assurance also depends crucially on qualitative analysis.

In confirming extra-functional properties, it is vital to employ complementary simulation and analysis techniques. Here, simulation entails dynamic analysis, as contrasted with the static nature of analysis *per* se. Their mutually reinforcing attributes are outlined in Table 2. Necessarily, all assessments must use precise and consistent architectural representations. Then, the corroboration results obtained from static analysis and simulation must be established. Finally, the analytical models must be consistent with the finalized architecture definition.

.	ANALYSIS	SIMULATION
SCOPE	General Conclusions	Particular Conclusions
ORIENTATION	Equivalence Classes (Breadth)	Problematic Scenarios (Depth)
DOMAIN	Encompassing Properties	Selective Subset of Behaviors
KEY	Tractable yet Admissable Model Simplifications	Representative Scenario Selections
MECHANISM	Reasoning/Consequences	Stimulation/Observations
MODE	Static/Detached	Dynamic/Tangible
CLOSURE	Deductive	Inductive

Table 2 - Attributes of Analysis and Simulation

2.3 Specification-Driven Prototyping

The importance of the dynamic analysis of a developmental infrastructure architecture has been noted. This refers especially to the capacity to examine complex behaviors and models with selective high fidelity. Aside from detailed models, higher fidelity may involve an absolute timebase, the modeling of concurrency, or realistic scenarios. As noted in Table 2, problematic scenarios like the transient behavior exhibited during fault handling can be investigated through simulation, or prototype execution. Comparable static analysis is in general intractable. Since capabilities like timely and assured fault handling are essential to safety-critical systems, prototype execution is a vital architecting tool for ensuring reliability and safety from the outset.

The customary forms of software-implemented prototypes are the throwaway and the evolutionary approaches. Both approaches suffer from a lack of inherent discipline and focus. This renders their use for architecting critical systems problematic. The throwaway prototype has to be reversed engineered to recover the implicit semantics from code to propagate into actual product; this is a nebulous and error-prone task. Evolutionary prototypes tend to diverge from strictly relevant features and to embody dubious structure on which to base a development product.

Specification-driven prototyping was developed expressly to aid in the architecting of safety-critical systems. It overcomes problems associated with both throwaway and evolutionary prototyping, and provides a stable and rigorous basis for overall software system development. In specification-driven prototyping, neither the prototype nor its specification is thrown away. They are useful throughout system development, provided they are kept consistent with related aspects of the developmental software system. Further, the prototype is not itself evolved; only its specification is. As depicted in Figure 1, all prototype definition and development occurs at the specification level. The prototype is merely kept consistent with its specification. Hence, the prototype is not apt to diverge from the system requirements, and its complete semantics are always available in a version that has been partially verified through prototype execution.

Property	Realization	Role
Communicating	External events	Message passing Notification Request Timeout
Extended	Tokens	Local state data Message parameters
	Timing	Process duration Scheduling times Transmission delays
	Stochastics	Timing variation Demand variability Stochastic decisions
Finite-State Machine	Statechart subgraph	Active objects

Table 3. Higher-Level Statechart Expressiveness

2.3.1 Payoff from Specification-Driven Prototyping - Ultimately, appropriate information from the prototype specification is propagated to product specifications. Such information includes global concurrency logic, exact timing parameters with tolerances, and quantitative parameters such as sizing values. Unfortunately, this kind of information is often conservatively estimated in early-on product specifications, or acknowledged only by "TBDs." If such information is not cogently stipulated, the basis for subsequent development is questionable. Ill-informed or defaulted specification entries tend to yield component/sizing mismatches, performance deficiencies, trial-and-error development, and undue system complexity.

In summary, the motivation for specification-driven prototyping is to establish global concurrency logic and quantitative parameter values for the infrastructure architecture. The intent is not only to eliminate uncertainty, but also to ensure a more balanced, robust, and economical design. These attributes translate into a system implementation that has: assured real-time performance under worst-case conditions; minimal spurious fault alarms and service outages; freedom from disparate bottlenecks and surplus capacity; and in general stable parameter values.

2.3.2 Higher-Level Statecharts (HLSs) - HLSs were developed expressly to support specification-driven prototyping[2]. They were needed because of the lack of any other language with suitable expressiveness at that time. Previously, a kind of higher-level Petri net, a predicate-transition network, had been used, but it did not provide desired modeling construct modularity. Since a subgraph in a basic statechart constitutes a finite-state machine (FSM), statecharts possess the relevant modularity property. They model communication among FSMs through events issued across subgraph boundaries. These circumstances prompted innovations to extend the expressiveness of basic statecharts to that equivalent to predicate-transition networks, or the development of HLSs. So features like circulating tokens and complex transition rule syntax were added to basic statecharts to define HLSs.

HLSs may be thought of as *scalable* communicating extended finite-state machines (CEFSMs), with correspondences as seen in Table 2. The inherent scalability applies to tokens as well as to subgraphs. Tokens are based on (passive) data types, and subgraphs are defined as (active) process types. Prefix-dot notation is used to denote scalability in both cases. The result is arbitrary scalability with no changes at all to transition rules.

While of the same form as used for basic statecharts, the syntax for HLS transition rules is appreciably more complex. Augmented first-order logic is used to express HLS transition rules, with additional constructs to represent absolute timing and stochastic effects. Some other aspects of HLS transition rule notation are:

- Distinction between Logical- and Operational-ANDs;
- Distinction between Logical Exclusive-OR and Operational Exclusive-OR;
- Notation for Token Migration between subgraph Nodes;
- Compound Action-Parts;
- In-line Comments in Action-Parts.

Overall, HLSs enable a coherent form of nested abstractions, where the HLS itself represents an encompassing concurrency model. HLS subgraphs denote interacting process abstractions, which in turn operate on instances of various token types, or data abstractions. Consequently, state data is captured at three different levels: global, process, and data token levels. This multi-level state characterization coincides well with the aforementioned control state decomposition methodology, with its emphasis on system state and operational modes. And the top-level states and modes correspond explicitly with the models used for static analysis and product specification provisions.

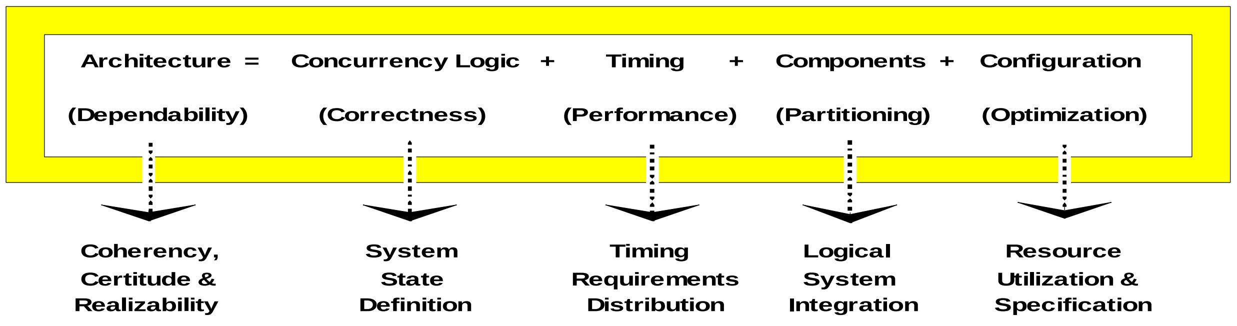
While HLSs have been applied extensively on a manual basis, their precise definition has not been completed. Accordingly, their precise mapping to an executable prototype has not been pursued. Consequently, such mappings have also been performed on a manual basis. Nevertheless, in numerous instances, successful prototypes have been developed, and the HLSs have been very incisive in informing prototype refinements.

3 PROTOTYPING METHODOLOGY

The prototyping methodology is summarized in Figure 3, where the same infrastructure architecture model, and hence prototype specification, are evolved over four stages of elaboration and assessment. Overall, the intent is to rigorously define and verify an infrastructure architecture with confirmed dependability properties like safety. The initial prototyping stage investigates the correctness of concurrency logic for the global management of the software system. Here, HLS process types correspond to logical elements in the nascent architecture. This activity serves to reveal many HLS specification deficiencies, and ultimately, to establish precise definition and management of overall system state(s).

The second stage of prototyping dynamically examines the absolute timing of interleaved processes in a distributed system, based on applied stimuli and the affected HLS transition rules. Here, the transition rules are expanded to include timing terms and possibly stochastic variations in prototype operation. The emphasis is on confirming real-time response and performance.

The third stage focuses on physical component partitioning and interfaces, which tend to associate with process type boundaries. Distribution of timing requirements and tolerances are allocated among components as well. Refined estimates for the applications architecture needs are introduced at this stage, which corresponds to the System Architecture Integration block in Figure 2. The application demands are merely simulated as dummy loads.



- Same Architecture Model(s) Evolved over the 4 Stages
- Outputs of Logic & Quantitative Parameters for Specifications

Figure 3 - Infrastructure Architecting Progression

The final stage is an optimization effort where the specific numbers of various component types and their respective quantitative parameters can be set to best overall advantage. This involves the use of stochastic simulation and objective functions. For example, a genetic algorithm has been used with stochastic simulation to optimize component counts and parameters[3].

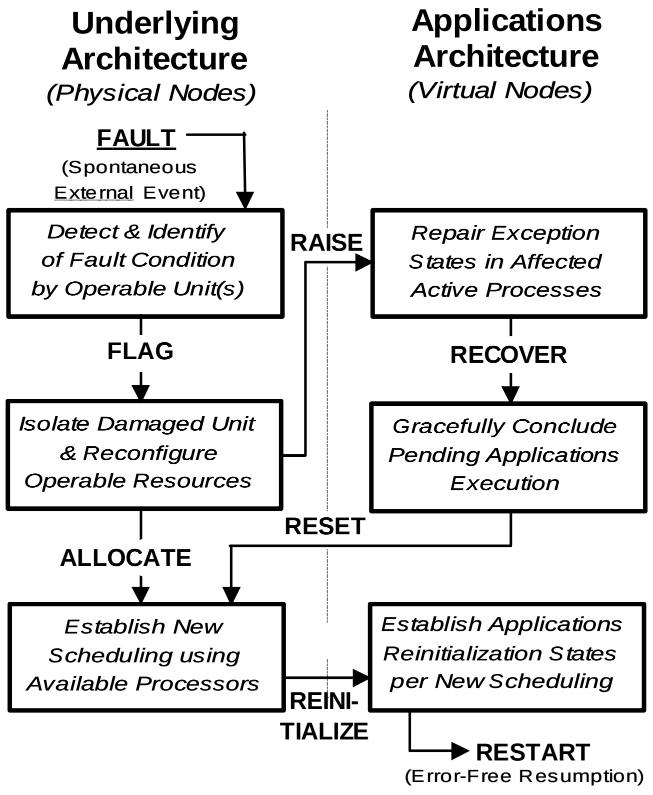
This evolution of the specification-based prototype imparts precision and certitude, and hence dependability, to crucial aspects of the architecting process. Prototype execution also stimulates conceptual and usage insights. When appropriate, the first stage of prototyping, concurrency logic correctness, can be skipped. This does not affect the prototype specification, but it greatly simplifies its implementation. Even though the process remains a manual one, the benefits for crucial aspects of critical systems are seen as well worthwhile for orderly development and economical design.

3.1 Prototyping Experience

Two categories of experience with specification-driven prototyping have accrued: that derived during architecting activities, and that obtained in supporting physical system development. Architecting experience has in part accrued in the modeling of rate monotonic scheduling, Ada virtual nodes and remote rendezvous, the Pilot's Associate processing concept, and the optimization of an on-line transaction processing system[3].

Figure 4 depicts some results of prototype execution for the Ada virtual nodes concept. Here, the failure of one of the physical nodes has been simulated. Detection of the hardware fault has then prompted processor reconfiguration and application restart. The restart is based on given applications priorities and the availability of standby virtual nodes at operable physical nodes. The diagram also shows the interplay between the underlying and the applications architectures in the prioritized restoration of services. HLS events are shown in bold font on the interconnection arcs.

Two cases of prototyping support for system development are notable. First, a subtle quad channel synchronization problem in a physical system was corrected with about three hours of prototype experimentation. Previously, a week of physical system testing had failed to even diagnose the problem. Second, an intermittent synchronization dropout by one particular computational channel was quickly diagnosed with the use of the prototype. Previously, two weeks of system testing had failed to disclose the source of the problem. Derivatives of a prototype have also been used successfully for real-time system execution monitors.



Underlying Architecture
(Physical Nodes)
Applications Architecture
(Virtual Nodes)
FAULT
(Spontaneous External Event)
Detect & Identify of Fault Condition by Operable Unit(s)
RAISE
Repair Exception States in Affected Active Processes
FLAG
RECOVER
Isolate Damaged Unit & Reconfigure Operable Resources
Gracefully Conclude Pending Applications Execution
RESET
ALLOCATE
Establish New Scheduling using Available Processors
Establish Applications Reinitialization States per New Scheduling
REINI-TIALIZE
RESTART
(Error-Free Resumption)

Figure 4. Prototype Execution of Fault Handling

REFERENCES

1. MULCARE, D.B. *et al.* 1984. Analytical design and assurance of digital flight control system structure. *AIAA Journal of Guidance, Control, and Dynamics*, May-June 1984.
2. MULCARE, D.B. 1993. Ada multitasking prototyping using higher-level statecharts, Tutorial at TRI-Ada '93.
3.MULCARE, D.B. 1996. System-level optimization of architectural performance under varying service demands. *9th International IEEE Symposium and Workshop on Engineering of Computer-Based Systems.*