A Uniform Software Architecture for Cooperation, Reliability and Reconfiguration of Autonomous Decentralized Systems *

Alvin S. Lim
Computer Sciences Department
Clark Atlanta University
Atlanta, GA 30314.

Abstract

Large and complex decentralized systems with autonomous components often require appropriate mechanisms for cooperation, reliability, and dynamic reconfiguration. It is crucial that these mechanisms allow the system to execute continuously for a long period of time with minimal disruption, despite failure or reconfiguration of some components. They should also allow various components to interact in complex ways. We develop a uniform software architecture for implementing these mechanisms using generalized techniques and common functionalities. These mechanisms also support a variety of application-dependent policies.

1 Introduction

There is an increasing demands for large and complex decentralized systems, such as computer-aided automated manufacturing [7, 12, 23], network service applications, and process control [24]. However, the development and execution control of these applications still pose many challenging and difficult problems [9]. To design an appropriate software and system environment for developing and executing these applications, we re-examine the salient characteristics of these applications.

Many of these large applications are distributed, execute for a long period of time and may interact in complex ways. They may also involve many interacting components and are subject to frequent independent failures and regular dynamic reconfiguration for scheduled restructuring, replacement of old modules or installation of new modules.

Consider an automated manufacturing system that requires coordination of several workstations and material handling robots to produce a family of products from streams of raw material. Systems like this pose many challenging problems. First, concurrent operations may be non-serializable. Operations of the workstations and robot movement must be synchronized in complex ways to avoid interference and enforce cooperation. Second, backward recovery may be impossible if failure involves non-recoverable physical effects; forward recovery may be the only option. Third, the configuration of these systems often changes over time. For manufacturing facilities to maintain the competitive edge, it is important to provide the capability for maintaining continuous operation during dynamic reconfiguration of a plant to meet the rapid fluctuation in market demands for new products. Configuration change may also result from regularly scheduled maintenance and from periodic upgrades because of improvement in technology or design. Conventional synchronization and recovery schemes, as well as existing automated manufacturing software, provide poor support for reconfiguration.

A major problem in these applications is maintaining consistency in the presence of concurrency, failure and reconfiguration. Atomic transactions are commonly used to simplify the management of concurrency and failure by preserving serializability and failure atomicity. The drawback of preserving the serializability and failure atomicity properties is that they restrict the types of synchronization that can be specified. There is a mismatch between the properties of atomic transactions and the characteristics of general distributed systems. Instead of making incremental extensions to transactions, we explore a new approach based on a general synchronization model of process interaction. It allows software designers to specify general interactive behavior. We introduce a novel set of correctness conditions for ensuring consistency that do not require applications to be serializable or atomic. These conditions allow semantics of applications to be exploited to enhance concurrency and the control of

*Research supported in part by ARO Grant DAAL03-92-G-0377.

recovery and dynamic reconfiguration.

We have developed a uniform framework based on state machines that is appropriate for specifying and modeling the partial-order behavior of distributed processes and analyzing synchronization problems. It also lends itself to automatic analysis of global consistency that must be maintained during failure recovery [16] and dynamic reconfiguration [15]. We model distributed processes as finite state machines (FSM's). Each process executes autonomously but may interact with others. To help designers specify the behavior of groups of processes, we propose a hierarchical FSM model that includes mechanisms for abstraction and composition. This hierarchical model enables us to control the runtime behavior of processes in a modular manner. The abstraction of a composite machine limits the scope of analysis and projects away uninteresting or illegal states. This makes the system scalable.

In this paper, we however restrict our domain of applications to finite-state processes and exclude some systems, as described in [22], which have infinite state behavior, such as unbounded FIFO channels. Extending our model to these applications, while possible, is beyond the scope of this paper. The FMS model is useful for many distributed applications such as automated manufacturing [7]. We will compare our system with other works only in the context of finite-state processes.

In Section 2, we present a manufacturing example that motivates our work. Section 3 then describes the software architecture for supporting such manufacturing applications. Section 4 describes how designers may specify a complex application cooperation, failure recovery actions and reconfiguration methods. The specification is supplied to a composite FSM compiler described in Section 5. The results of the compilation and analysis is supplied to a runtime control manager, described in Section 6, which uses the information to control synchronization, recovery and reconfiguration. We compare our research with other work in Section 7. Finally, in Section 8, we close with some concluding remarks and summary.

2 Motivating Example

Consider the following example of a manufacturing cell (Figure 1) that consists of a cutter, two milling machines, an assembler, and a robot.

A workpiece is first cut, then each piece is milled according to different specifications, and finally the pieces are assembled together. The robot is responsible for transporting the workpieces from one machine to another, constrained by the timing of the operations and conditions of the machines. Each machine operates for a long period in a cyclic manner, i.e. repeatedly waiting then performing its function.

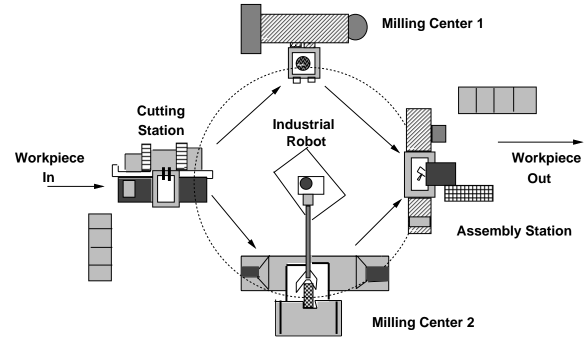

Figure 1: A Manufacturing Cell

This example will illustrate our synchronization, recovery, and reconfiguration goals. First, we need to synchronize the cyclic operation of the individual machines to obtain correct behavior. For example, the cutter must have finished before the robot tries to grab a workpiece, and the milling machine must be idle before the robot tries to deliver one. Next, we must provide for the inevitable failures or exceptions that will arise during an operation. If a tool breaks during a milling operation, we must recover the group of machines to an acceptable state, which may mean discarding the otherwise unaffected piece being processed by the second milling machine. Finally, and critically in the automated manufacturing domain, the programs controlling each machine must be replaceable without shutting down the whole group of machines. Such program changes are common for many reasons: improvements in production processes, engineering design changes, or quick switches to producing a different part in a family of related parts, etc.

Today's automated manufacturing software is poor at reconfiguration without downtime. Approaches in [27] require the manufacturing system to be stopped for newly generated software to be installed and initiated. They do not address the problem of maintaining consistency of other components that interact with the changed components. All the changes just mentioned must be managed in the context of a larger factory, where our example cell is just one part of a production system and the workpiece is a part of a larger job.

3 Software Architecture

The uniform software architecture consists of two subsystems: static and dynamic (Figure 2).

3.1 Static Subsystem

The static subsystem consists of (1) a compiler that takes the specification of a decentralized system and

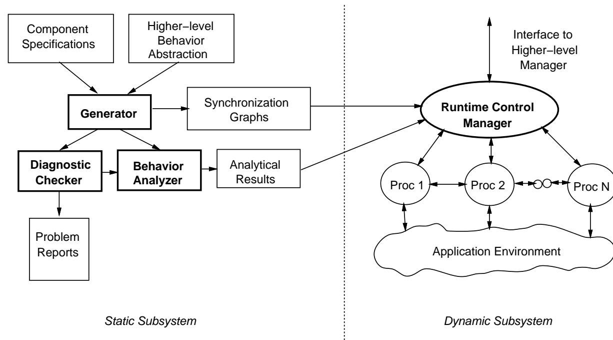

Figure 2: Uniform Software Architecture

produce intermediate results for analysis and runtime control, (2) a diagnostic checker that ensures correctness of the specification, and (3) a behavior analyzer that verifies important liveness properties and produces analytical results for potential failure recovery and dynamic reconfiguration.

The behaviors of each process may be specified and analyzed independently of the distributed application. A programmer supplies a specification of an application to the compiler that produces intermediate results to the diagnostic checker which checks for syntactic correctness, overall reachability of all states, and feasibility of each abstract behavior. Problems are reported to the programmer for correction. The behavior analyzer automatically verifies the potential for deadlock, livelock and starvation. While deadlocks are usually easily removed by specifying additional constraints, livelock and starvation are probabilistic properties which are not easily removed by modifying the specification. Starvations and livelocks are better handled by dynamically detecting them and recovering the affected processes.

3.2 Dynamic Subsystem

The dynamic subsystem consists of the runtime control manager that takes the results from the static subsystem to control synchronization of processes and maintains consistency during failure recovery and dynamic reconfiguration. However, each process executes autonomously to control equipment or physical machines in a manufacturing cell or material handling system and may interact through messages with one another to perform a manufacturing task. For example, the milling machine controller may communicate with the robot and workpiece controller to remove milled workpiece [7]. In an hierarchical application structure, the manager coordinates with other higher-level and peer managers. Managers are not customized for each application. Instead, they are driven by the output of the generator's behavior analyzer.

3.3 Policies vs. Mechanisms

We enforce a separation of policies from mechanisms. The well-known separation of policy and mechanism [13] has not been previously applied to synchronization, reliability and reconfiguration of decentralized systems. Policies that software designers may specify through the model is independent of the runtime mechanism that dynamically enforces an application behavior specified in the model and provides automated failure recovery as well as reconfiguration of running applications. This paper focuses mainly on the runtime mechanism. Software designers may create processes for enforcing policies for optimal task scheduling [25] or selection of the best (set of) sequences of operations by a product planner system [23]. However, techniques for policies decision making are beyond the scope of this paper.

4 Specification

Developers may specify autonomous decentralized systems in a framework based on a hierarchical finite-state machine model. Autonomous decentralized systems are usually complex and consist of many independent but cooperating components. We model the behavior of each process by a finite-state machine, called a *basic machine*. A *basic state* represents a clean, internally consistent state of the process. We do not intend an FSM to model all possible program states; *only* those states representing important points in the process execution, such as synchronization points, consistent checkpoints, a unit of work, etc. The programmer provides the many-to-one mapping from program states to states in the FSM's. A *basic transition* represents a sequence of internal operations that move the machine from one basic state to another. Our interpretation of finite-state machines is similar to [14] in that states are clean-points and transitions represent operations requiring some time. Software designers define operations of basic machines without specifying synchronization with other basic machines. Each basic machine executes independently but may interact with others. For example, the processes that control the manufacturing cell in Figure 1 execute autonomously. The behavior of the cutter, milling centers, welder, workpieces and the robot are controlled by the basic machines shown in Figure 3.

A Cartesian product of a group of basic machines is called a *product machine*. It consists of product states and transitions which describes all possible behavior of the group. A *product state* is a tuple of concurrent basic states. A *product transition* moves from one product state to another. (An equivalent asynchronous model for specifying constraints on concurrent execution is described in [15]. All concepts discussed here

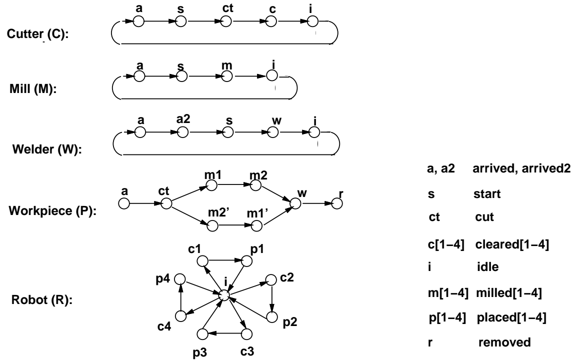

Figure 3: Basic Machines of the Manufacturing Cell

are also applicable in the asynchronous model.)

4.1 Synchronization

We allow developers to constrain their systems by specifying the abstract behavior of a group of processes using composite finite-state machines. A *composite machine* is specified using abstract states and transitions and constraints on product states and transitions specified using first-order logical expressions over basic machine states an transitions. Abstract states and transitions map to sets of product states and transitions. Software designers can erase (i.e. forbid) any product states and transitions to express arbitrary synchronization constraints. After removing the forbidden states and transitions, the resulting FSM is called a *restricted product machine*. Restricted product machines are generated and supplied to a runtime control manager that enforces synchronization (and manages recovery and reconfiguration) of basic machines (discussed in Section 6).

For example, to ensure that the cutter has completed before the robot tries to grab a workpiece, we add the following constraint on states reachable from all composite state. ($*$ means that the constraint applies to all composite states.)

$$* : \qquad R.c1 \Rightarrow C.ct$$

4.2 Failure Recovery

In the preceding, we described a normal model, with normal states and transitions, which characterizes the behavior of applications during normal execution, i.e. in the absence of failure. We now augment the FSM model to characterize behavior involving failure and recovery. To the states of the normal model, we add a set of *failure states*. We consider only fail-stop failures. There may be other classes of failure, but we only deal with these. Software designers may define the failure states for each individual basic machine (in addition to some system default failure states). A failure product state is a product state that contains at least one failure basic state. We also add *recovery transitions* from failure states to normal states. Recovery transitions represent operations that must be implemented either by the basic machine designer or the runtime control manager. For a group of basic machines, we can generate the appropriate *recovery path*, defined as a sequence of recovery transitions from a failure state to a normal product state called a *recovery state*.

Consider the recovery of a milling operation failure in the manufacturing cell. The programmer specifies failure states for individual basic machines, e.g. Φ for Mill and Ψ for Workpiece. Mill (Workpiece) enters Φ (Ψ) when failure occurs at any of the states: a, s, or m ($ct, m1, m2, m1'$, or $m2'$). He may also specify the recovery operations: $\Phi \rightarrow M.a$ and $\Psi \rightarrow W.a$. When a tool breaks during a milling operation, the Mill basic machine enters a failure state Φ and the Workpiece basic machine enters a failure state Ψ. The runtime manager then determines the best recovery path and forces the individual processes through the recovery action.

4.3 Reconfiguration Methods

To modify the configuration of an application, a software designer or system administrator supply a set of reconfiguration transitions or the desired configurations. A *reconfiguration transition* always moves from a product state in one configuration to a product state in another configuration. Every state in the current configuration is the source of a graph sinked at a state in the desired configuration. This graph is a subgraph of the Cartesian product of reconfiguration transitions. A reconfiguration sequence maps a state in the current configuration to a state in the new configuration.

During reconfiguration, processes may be added and there may be more concurrent processes than either the current or target configuration contains. These concurrent processes may interfere with one another, although there is no constraint restricting them in either the current or target configuration. If interference is possible, the system administrator must supply additional synchronization constraints on their behavior. These additional synchronization constraints are called *transient constraints*. Transient constraints apply only during dynamic reconfiguration and remove disallowed reconfiguration transitions. For example, in the manufacturing application (Figure 1), to prevent reconfiguration of Milling Machine 2 while Workpiece is at ct or $m1$, a transient constraint may remove those reconfiguration transitions leaving product states in which Workpiece is in state ct or $m1$.

4.4 Hierarchical Composition

To enhance scalability, we allow machines to be hierarchically composed. Each composite machine may

be used as a basic machine to construct higher level composite machines by a higher level runtime control manager where only composite states and transitions are visible to composite machines at higher level. (Individual basic machine states and transitions are hidden from higher level composite machines.) Each lower level composite machine is controlled independently by its runtime manager. Figure 4 shows a two-level hierarchy of composite machines. Only level A composite

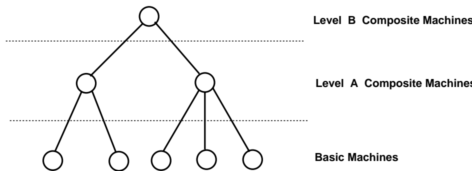

Figure 4: A Hierarchy of Composite Machines

states and transitions are visible to the level B composite machines; lower-level basic machine states and transitions are transparent to the level B composite machines. Composite machines at level B view composite transitions (states) at level A as basic transitions (states) although they may represent sequences of lower-level basic transitions (sets of lower-level basic states).

The capability to specify hierarchical composite machines assists application programmers in building large and complex distributed systems by grouping together smaller modules at each level. Each composite machine defines a limited scope in which the behavior of a group of machines can be independently analyzed and controlled. We restrict the scope of composite machines at each level to enable analysis and control of hierarchical composite machines to be manageable.

5 Composite FSM Compiler

The composite FSM compiler accepts programmer specifications of basic and composite machines. It first makes several diagnostic checks:

- syntactic correctness
- overall graph connectivity of each machine
- feasibility of each composite transition

Invariant constraints specified on product states and transitions are checked to verify that they are consistent with each other. Complete graph connectivity is important since some product machine states will be unreachable if there are disjoint connected components in the restricted product machine graph. In addition, all specified composite transitions must be feasible, i.e. there must be some product states in the destination composite state that are reachable from at least one product state in the source composite state. Problems are reported to the programmer for correction.

If the input is acceptable, the compiler then analyzes the specification to identify:

- sets of legal product transitions for each composite state and transition
- deadlocked product states
- potential livelocked and starved product states

The compiler determines regions in the restricted product machine that lead to deadlock or possible livelock or starvation. Each product state and transition is labeled to indicate its legality, the composite states that are reachable, the composite states it belongs to (if any), and the composite states that inhibit it. Product states are also labeled if they are in a region that may deadlock, potentially livelock, or cause starvation. The compiler computes a path from each failure and problem state to acceptable states. These paths can include recovery actions (such as basic machine reset).

6 Runtime Control

After compilation and analysis, the result is supplied to a runtime control manager. The manager controls scheduling, enforces synchronization control, implements recovery, manages reconfiguration, and takes the role of a basic machine when the composite is used in a higher-level group. Managers are not customized for each composite machine. Instead, the separation of policy and mechanism lets them be driven by the output of the compiler's behavior analyzer.

6.1 Synchronization

The manager controls synchronization by repeatedly listening and serving requests from all basic machines. When each basic machine is instantiated, it first calls `announce` to announce its initial state. To request permission to move to a destination state, it will then call `request_permit` (implemented as a remote procedure call to the manager). The manager examines the restricted product machine graph to determine if the requested transition violates synchronization constraints or will lead to some problems. To improve concurrency, two or more product transitions are permitted to execute concurrently at a product state S if there is a legal path from S for *every* permutation of those transitions. If acceptable, the manager authorizes the transition by returning `GRANTED` to the pending `request_permit` call. Otherwise, it returns `HOLD`. The basic machine may then decide to make an alternative transition or wait for the transition to become acceptable by calling `block_request`. The manager then puts the request in

a pending list and returns GRANTED when the transition becomes acceptable.

Consider the manufacturing cell, where the Welder W is at state s (a welding operation is in progress) and the Robot R is in state $c3$ (holding a workpiece). R then request permission to move to state $p3$ (place workpiece in Welder). The manager consults the restricted product machine graph that indicates that the target product state is illegal. The manager then returns HOLD to inform R to wait. Meanwhile, W completes the welding operation and move to state i. At that point, the manager returns GRANTED to allow R move to state $p3$.

6.2 Recovery

If the composite machine violates its concurrency and sequencing constraints for any reason, the manager initiates exception mode. Examples of such violations include failure of a basic machine action and unspecified basic transition. The manager may also be forced into exception mode by a manager of a higher-level composite. When a basic machine detects a failure, it may notify the manager of the failure immediately by calling announce. However, a process may fail without informing the manager of the failure. When the manager has not received any request from a process for an extended time period, the manager may query the process status by calling the function query_status.

If a basic machine announces a failure or returns a failure status to the query, the manager forces the basic machine from an autonomous mode to an exception mode by calling the function force_mode with the mode EXCEPTION. From the restricted product machine, the manager determines the appropriate recovery path. The manager then calls mandatory_transition to force the affected basic machines through the recovery path. On receiving a mandatory_transition notification from the manager, the basic machine tries to comply and responds by returning the status OKAY if the mandatory transition succeeds, and FAIL otherwise. If the response is FAIL, the manager may attempt alternate recovery paths or report the failure to the system administrator. After restoring the application to a recovery state, the manager resumes normal operation by calling force_mode to set all basic machines back to AUTONOMOUS mode and then grants permission for outstanding legal transitions.

6.3 Dynamic Reconfiguration

To reconfigure an application, a system administrator supplies a set of reconfiguration transitions and transient constraints to a compiler that generates a reconfiguration graph. The runtime control manager determines the affected basic machines involved in both the partial recovery path in the current configuration and the reconfiguration transitions. It then forces all affected basic machines from autonomous mode to exception mode using the function force_mode. The manager then calls mandatory_transition to force the basic machines through the partial recovery path to a product state where reconfiguration transitions may begin. Reconfiguration transitions in the reconfiguration path are then executed by a (supervisor) process that controls basic machines and has the privileges for killing and starting basic machines. This role could be assumed by the manager itself. A reconfiguration transition that replaces, add, or removes processes may be implemented with system-provided primitive mechanisms: save (restore) states, and create (remove) and execute basic machines. After successfully executing the reconfiguration transitions, to a state in the new configuration, the manager then calls mandatory_transition to force basic machines through the second recovery path to restore global consistency in the new configuration. The manager calls force_mode to force the basic machines back into autonomous mode.

7 Comparison with Other Work

There are only a few existing systems that provide uniform software architecture for controlling correct synchronization, reliability and reconfiguration in a unified way. Two such systems [17, 18] are based on transactions and have the restrictions discussed in Section 1. The first is the Argus transaction mechanism [17] with the extension of dynamic reconfiguration mechanism by Bloom [5]. Argus, however, does not provide support for incorporating new recovery techniques, such as compensation used in forward recovery. Furthermore, the reconfiguration mechanism permits only new configuration with behavior that is identical to that of the old configuration. In flexible manufacturing systems, the behavior of a new configuration often differs from that of the old configuration. Conic [18] gives only a partial framework; providing support for dynamic reconfiguration but little support for recovery management. Durra [4] and Polylith [21] deal primarily with the problem of structural reconfiguration and the mechanisms for capturing and restoring states, but did not deal with the problem of maintaining state consistency. In [8], Hailpern and Kaiser mainly address the problem of maintaining type consistency during dynamic reconfiguration. There are several other software architectures that focus mainly on specification and/or reliability [29, 19]. Dynamic reconfiguration, though an important aspect of autonomous decentralized systems, was not addressed.

Other work has concentrated on allowing designers to customize a limited set of application-specific techniques only for recovery management [10, 26, 17].

These systems provide the basic mechanisms for supporting transactions, e.g. transaction creation and commit, logging, and recovery management. QuickSilver [10] exposes a set of primitives that is at a lower level than most other systems, such as Camelot [26] and Argus [17], and permits servers to implement their own recoverable storage and log recovery. Extended transactions, such as atomic abstract data types [28] and ACTA [6], require transactions to be serializable after non-dependent operations are commuted. The burden of analyzing commutativity (or dependency) between operations is placed on the programmers. Optimistic approaches [11] also require committed transactions to be serializable. Cooperative transactions [3] extend basic nested transactions, but still require the partial order of lower level nested transactions to be equivalent to a total order of operations invoked by subtransactions of the cooperating transaction. In contrast, our approach does not require concurrent processes to be serializable. They may involve non-commutatable interactions. When failure occurs, processes may recover to some intermediate execution points [16]. Consistency is preserved by automatically checking for all operations dependent on those recovered operations and recovering them.

Recently, hybrid automata [20, 2] have been investigated as a viable model for hybrid systems (e.g. physical plant control) which contains two distinct types of systems – continuous and discrete-state – that interacts with each other. Hybrid automata contain real-valued variables whose behavior at each state is governed by a set of differential equations. Although we have used a simple FSM model in this paper, the architecture can be extended to manage plant control based on hybrid automata.

Our main focus here is providing a uniform software architecture for autonomous decentralized applications that has three important characteristics. First, we have separated a common control mechanism from the recovery and reconfiguration policies that ensure preservation of consistency. This allows uniform control of the normal execution of cooperative operations as well as different techniques for failure recovery and reconfiguration. Processes need not be prevented from interacting with others within well-defined boundaries (e.g. begin and commit transaction). Furthermore, in the presence of failure or reconfiguration, processes may be recovered to some intermediate execution points. Second, it automatically determines those processes that are unaffected by the failure or reconfiguration and allows them to continue their normal operations. For example, a machine in a manufacturing cell can be upgraded or replaced without shutting down the entire group of machines. In the presence of failure and reconfiguration, the system maintains consistency by automatically analyzing dependency between operations. Designers are relieved of the burden of checking for consistency. Third, synchronization, recovery, and dynamic reconfiguration is controlled in a unified way based on an open model that allows combination of different recovery or reconfiguration techniques to be implemented [16].

Although our current approach is applicable only to applications, such as automated manufacturing, where each process and group has well-defined behavior and are finite-state, we believe our software architecture can be extended to open systems. An attempt at accomplishing similar goals has been demonstrated in [1].

8 Conclusions

We have described a uniform software architecture with common functionalities for supporting cooperation, failure recovery and dynamic reconfiguration which are important in many large autonomous decentralized systems, such as automated manufacturing. A runtime manager provides common, simple, and efficient interface to a group of autonomous processes that enables them to cooperate in complex ways without being restricted by recovery or reconfiguration mechanisms. Each group of processes is controlled by its runtime manager independently. To enhance autonomy, software designer appropriately design group of processes hierarchically. When failure occurs, affected process may recover from the failure while the runtime manager transparently maintains consistency with other cooperating processes. During reconfiguration, there is minimal disruption of other processes unaffected by the reconfiguration, whereby the runtime manager transparently analyzes dependencies and determines the set of affected processes. By separating the mechanisms from the policies, we allow different policies to be implemented, including those that exploit application semantic to improve concurrency, recovery and reconfiguration.

References

[1] Abadi, M. and L. Lamport, "Open Systems in TLA," *Proc. 13th ACM Symp. on Principles of Distributed Computing*, Aug. 1994, pp. 81-90.

[2] Alur, R., C. Courcoubetis, T. Henzinger, P. Ho, "Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems," *Proc. Workshop on Theory of Hybrid Systems*, Lecture Notes in Computer Science 600, pp. 209-229, Springer-Verlag, 1991.

[3] Bancilhon, F., W. Kim, H. F. Korth, "A Model of CAD Transactions," *Proc. of the 11th Int. Conf. on Very Large Data Bases*, 1985, pp. 25-33.

[4] Barbacci, M. R., C. B. Weinstock, J. M. Wing, "Programming at the Processor-Memory-Switch Level," *Proc. 10th Int. Conf. on Software Engineering*, Singapore, April 1988.

[5] Bloom, T., "Dynamic Module Replacement in a Distributed System," Technical Report MIT/LCS/TR-303, *MIT Laboratory for Computer Science*, March 1983.

[6] Chrysanthis, P. K. and K. Ramamritham, "ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior," *Proc. of the ACM-SIGMOD Conf. on Management of Data*, May 1990, pp. 194-203.

[7] Duffie, N. A., R. Chitturi, J. Mou, "Fault-tolerant Heterarchical Control of Heterogeneous Manufacturing System Entities," *J. of Manufacturing Systems*, Vol. 7, No. 4, 1988, pp.315-328.

[8] Hailpern, B. and G. E. Kaiser, "Dynamic Reconfiguration in an Object-Based Programming Language with Distributed Shared Data," *Proc. 11th Int. Conf. on Distributed Computing Systems*, May 1991, pp 73-80.

[9] Harel, David, "Biting the Silver Bullet," *IEEE Computer*, Vol. 25, No. 1, January 1992, pp. 8-20.

[10] Haskin, R., Y. Malachi, W. Sawdon, G. Chan, "Recovery Management in QuickSilver," *ACM Trans. on Computer Systems*, Vol. 6, No. 1, Feb. 1988, pp. 82-108.

[11] Herlihy, M., "Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types," *ACM Trans. on Database Systems*, Vol. 15, No. 1, March 1998, pp. 96-124.

[12] Jones, A. T. and C. R. McLean "A Proposed Hierarchical Control Model for Automated Manufacturing Systems," *J. of Manufacturing Systems*, Vol. 5, No. 1, 1986, pp.15-25.

[13] Jones, Anita K. and William A. Wulf, "Towards the Design of Secure Systems," *Software – Practice and Experience*, Vol. 5, 1975, pp.321-336.

[14] Lamport, Leslie, "Specifying Concurrent Program Modules," *ACM Trans. on Programming Languages and Systems*, Vol. 5, No. 2, April 1983, pp. 190-222.

[15] Lim, Alvin S., "A State Machine Approach to Reliable and Dynamically Reconfigurable Distributed Systems," Ph.D. Dissertation, Technical Report 1132, Department of Computer Sciences, University of Wisconsin-Madison, January 1993.

[16] Lim, Alvin S., Stuart A. Friedberg, "A State Machine Approach to Reliable Distributed Systems," *Proc. of the 11th IEEE Symp. on Reliable Distributed Systems*, Houston, Oct. 1992, pp. 204-212.

[17] Liskov, B., D. Curtis, P. Johnson, R. Scheifler, "Implementation of Argus," *Proc. of the 11th Symp. on Operating Systems*, Nov. 1987, pp. 111-122.

[18] Magee, J., J. Kramer, M. Sloman, "Constructing Distributed Systems in Conic," *IEEE Trans. on Software Engineering*, Vol. 15, No. 6, June 1989, pp. 663-675.

[19] Mishra, S., L. Peterson, and R. Schlichting, "Modularity in the Design and Implementation of Consul," *Int. Symp. on Autonmous Decentralized Systems*, Kawasaki, Japan, March 1993, pp. 376-382.

[20] Nerode, A. and W. Kohn, "Models for Hybrid Systems: Automata, Topologies, Controllability, Observability," *Proc. Workshop on Theory of Hybrid Systems*, Lecture Notes in Computer Science 600, pp. 317-356, Springer-Verlag, 1991.

[21] Purtilo, James M. and Christine R. Hofmeiser, "Dynamic Reconfiguration of Distributed Programs," *Proc. 11th Int. Conf. on Distributed Computing Systems*, May 1991, pp. 560-571.

[22] Ramadge, Peter J. G., W. Murray Wonham, "The Control of Discrete Event Systems," *Proc. of the IEEE*, Vol. 77, No. 1, Jan. 1989, pp. 81-98.

[23] Ranky, Paul G., *Computer Integrated Manufacturing*, Prentice-Hall Int., UK, Ltd., G. Britain, 1986.

[24] Schoeffler, J. D., "Distributed Computer Systems for Industrial Process Control," *IEEE Computer*, Vol. 17, No. 2, February 1984, pp. 11-18.

[25] Shaw, Michael J., "Dynamic Scheduling in Cellular Manufacturing Systems: A Framework for Networked Decision Making," *J. of Manufacturing Systems*, Vol. 7, No. 2, 1987, pp. 83-94.

[26] Spector, Alfred Z., et. al., "Camelot: A Distributed Transaction Facility for Mach and the Internet – An Interim Report," Technical Report CMU-CS-87-129, *Department of Computer Science, Carnegie Mellon University*, June 17, 1987.

[27] Weck, M., E. Kohen, "Configurable Control Software for FMS," *Software for Discrete Manufacturing: Proc. 6th Int. IFIP/IFAC Conf. of Software for Discrete Manufacturing*, Paris, June 1985, pp. 437-445.

[28] Weihl, W. E., “Commutativity-based Concurrency Control for Abstract Data Types,” IEEE Trans. on Computers, Vol. C-37, No.12, Dec 88, pp. 1488-1505.

[29] Yau, S. and G. Oh, “An Object-Oriented Approach to Software Development for Autonomous Decentralized Systems,” *Int. Symp. on Autonmous Decentralized Systems*, Japan, March 1993, pp. 37-43.