Controlled Emergent Adaptation: Management and Adaptation of Large Distributed Applications

Jonathan C. Hill

A Proposal for Dissertation Research
to Partially Fulfill the Degree
Doctor of Philosophy (Computer Science)
at the University of Virginia
School of Engineering and Applied Science
1 Introduction

We introduce a new control paradigm for scalable automated management and adaptation of distributed applications. Our paradigm combines control loop principles with a form of emergent actuation. It decreases management workload, increases the adaptability of applications, increases the scale of manageable application systems by orders of magnitude, and provides a new means for distributed application organization and design.

- **Control Loops** solidify the concept of an application management service layer, lying between the operating system and applications, that has been traditionally provided by the human operator.

- **Simple Extensibility** independently and asynchronously applies the expert management and adaptation technology of many developers to the same application system.

- **Controlled Emergent Adaptation** combines specified and emergent adaptation through a combination of top-down control and emergent action organization.

Three technology contributions drive our paradigm:

- **Cybernetic Nervous System**: A ‘nervous system’ consisting of multiple control loops is established alongside installed application components and a logical system model. The collection of control loops serves as a co-application—sensing, controlling, and actuating the management state of the application system.

- **Intention Council**: Regulation coordinates actions, preventing contradictions and inefficient ordering. Intention council performs this regulation while allowing controllers to act independently.

- **Multicast, Property-Bound Messaging**: The nervous system requires no ‘wiring’ of communication paths, as transmitters send messages to receivers over an efficient architecture that, to the external observer, resembles a collisionless, infinite-channelled bus. This greatly increases the scalability of the cybernetic nervous system paradigm by reducing its instance-design and actualization costs. It also allows run-time actuation binding, resulting in emergent actuation locality.

Together, these principles allow scalable automated network management and a new method of adaptive application system design. Actuations propagate into a network of actuation sites, binding where the appropriate local attributes are found. The result is a dynamically adaptive structure and organization of the system from local properties. The extent of emergent versus specified actuation and structure is a property of the formal specifications created by system administrators.
Figure 1 presents an overview of the paradigm’s operations at a point when a controller emits an action to enact on the system. The figure shows the path of control messages, beginning at controllers and destined for actuators. In existing systems, only the very top components (controllers) and bottom components (actuators) are present. Controllers and actuators communicate events to one another directly. Our primary contribution is an application management layer lying between controllers and actuators in the event flow-path. The management layer consists of two service layers: intention and actuation. These are conduits through which the regular communications between a controller and its actuators are conducted. The actual controllers and actuators are arbitrary; management and adaptation systems can apply their own controllers and actuators to the application system. Controllers and actuators can act independently of each other because the Intention Council Model resolves intention conflicts between controller commands. Then, Multicast, Property Bound Messaging binds intentions to actuation sites, instantiating the intentions wherever they are required. Dynamic resource management maintains a safe ordering of actuations whenever temporal ordering between actuations is not specified by controllers.

Application system management, in general, could benefit from the technology we propose. This benefit becomes necessary as the scale of application systems increases and traditional management techniques are rendered infeasible. Our primary goal is to test the value of the paradigm for larger application systems. End-to-end business systems, the national banking infrastructure, and the World Wide Web are examples of systems that could benefit from our paradigm. Attacks, management complexity, and design overhead increasingly threaten their viability. Our paradigm can reduce these costs with:

- increased flexibility of application systems to tolerate classes of anticipated and unanticipated faults,
- flexible, scalable management of common maintenance tasks, and
- reduced design cost through constrained self-organization.

This proposal outlines research on the utility of the paradigm for adaptation and management of large distributed applications. This proposal first discusses properties of application systems in Section 2. The paradigm for scalable configuration and adaptation is presented in Section 3. We derive an architecture for the paradigm in Section 4. Section 5 presents preliminary research studying the paradigm’s effectiveness. Finally, Section 6 proposes research to study the behavior of the paradigm, and characterize its effective scalability and manageability.
2 Distributed Application Systems

An application system consists of applications, operating together in a design to provide a total service.

*Definition 1: application system* a set of application resources and instances acting collectively to provide a function.

---

**FIGURE 1.** An overview of the paradigm: event flow from controllers to actuators.

- Control experts make abstract actuation decisions.
- Abstract actuations from controllers are called “Intentions.”
- Some intentions must wait or be rejected in arbitration, while others of higher priority are sent on to be actuated.
- Intentions are multicast to actuation sites with the correct local properties; where they are instantiated for local conditions and ordered for resource usage; resulting in safely ordered actuation.
A distributed application system involves application components potentially operating on more than one computing system. For example, the World Wide Web is a distributed application system. It consists of applications that serve web files (web servers) and clients that download the served content (web clients.) This system is wide-area and its components exist in various forms and technologies on various platforms. Independently, or in small groups, it represents a client-server distributed document technology. Together its overall application function is known as the World Wide Web. Other examples of distributed application systems include end-to-end business systems, software-driven telephony switching networks, military command and control systems, and financial information services [12].

2.1 Distinguishing Properties of Application Systems

Control of distributed application systems is complicated by their nature. There are several characteristics of application systems that distinguish them from typical homogeneous component-driven distributed systems:

- **High-Dimensioned, Extensible Actuation Space**
  Each application component has its own set of possible inputs thought of as a dimension in a discrete-valued space. With \( n \) applications each having \( m \) potential actuation variables, we have \( n \times m \) dimensions in an actuation space. Each dimension has at least two possible states. There are many constraints between dimensions’ values. For example, an application might be able to send email in an encrypted mode, but this requires that other email clients have installed components to decrypt the information. Hence, the potential actuation space of an application system is a highly constrained, high dimensioned space of possible configurations.

- **Non-interacting Applications Form Application Systems**
  Application systems need not consist of components that interact or share resources. They need only consist of applications that together provide a combined function. For example, computer applications may provide control over individual street-lights, but together provide the function of keeping occupied streets illuminated.

- **Multi-layered Conditions Affect Desired/Required Configuration**
  There are many layers of conditions affecting system configuration. For example, an application has its internal state. An operating system has its current state. Hardware is operating in a real-world environment with its own current conditions. This world might be subject to changing threats of physical or cyber-attack. Any of these levels of concern could require altered management and adaptation of the application system. This distinguishes the
required management from traditional (before QoS) operating system control, where management is of operating system resources based on operating resource conditions and a few ‘tweaking’ parameters. These properties distinguish application systems from traditional component systems. In traditional component distributed systems the actuation controls of components are often homogeneous, the affecting conditions limited in scale, and the modeled conditions limited to a few variables.

2.2 State-of-the-art in Distributed Application Systems

When state-of-the-art distributed application systems become sufficiently large, several negative properties emerge as a result of system complexity and the cost of adaptation:

- **Static**
  The size and complexity of the configuration space result in extensive administrative overhead. This overhead makes modification of the system expensive. In addition, the time required to make important adjustments is larger than the desired time to make a useful change. Therefore, application systems tend to be static in nature.

- **Fragile**
  These systems are prone to failure when adjusted without significant administrative investment.

- **Inefficient**
  The static and fragile nature of deployed application leads to conservative resource usage estimation, without the flexibility to better utilize resources under changing operating conditions. For example, security or reliability constraints for worst-case operating environment conditions may drive application distribution regardless of whether the operating environment is currently in a worst case scenario.

- **Vulnerable**
  Applications can compensate for vulnerability by responding to attacks or changing conditions prior to attacks. For example, an individual application on a stand-alone machine is deactivated or restarted by the operating system or a human operator. Static and fragile application systems are not safely altered by stand-alone operating systems or independently acting human operators. Actions on applications must be coordinated. While a distributed system has the potential to be less vulnerable to isolated attacks, it must correctly coordinate its components to avoid failure of its total function. Application systems, in general, do not have this coordination and are therefore more vulnerable rather than less vulnerable.
The resulting systems often contain archaic legacy components that are considered too costly to replace or adapt. If adaptability is provided for such systems, it is often an ad hoc collection of services that will only work with very specific, tailored application architectures.

The negative conditions in current application systems management motivates change. Increasing system scales exacerbate the above problems. If application systems are to be manageable at larger scales, new methods of management will be required. In addition, new methods should actively seek to avoid the re-emergence of these problems as application systems continue to grow.

3 Scalable, Automated Application Management

The life-cycle of deployed application systems includes installation, initialization, maintenance, adaptation to run-time environmental circumstances, and component replacement. We refer to activities such as these as management of an application system. Application system management must attempt to encompass all actuation potential of the system that lies above operating system implementation and outside of application implementation. Abstractly, this is a layer between the operating system and application system, as depicted in Figure 2. This layer is not the functionality of an operating system, nor the functionality of an application. This layer is not properly the application layer because it does not provide the services of an application, nor does the layer necessarily service a single type of application. It is not properly the operating system layer (unless it is subsumed in future systems), because it replaces many services that are expected to be provided by the operating system’s human operator. The cybernetic paradigm for application systems [23] formalizes the application management layer as an automated control layer.

‘Automated management’ means an increase in the abstraction of human input into the management of the application system. This also means the establishment of a proper software layer in the management layer of Figure 2. For
example, where a human may have once entered commands to install and run application components, automated management could allow a single command to perform the required set of tasks in the correct order and without further human input. Automated management does not preclude human input, yet it suggests that human input should be reduced or abstracted to a more appropriate level than is currently required. The purpose for this reduction/abstraction is three-fold:

1. Human input for redundant tasks is a waste of valuable and scarce human resources.
2. Human control of small tasks will not scale to large application systems.
3. Human input is slow and error prone, reducing the adaptation potential of application systems.

The goal of automated management is to allow an application system to perform its entire life-cycle with the greatest potential for its functionality. Automating the management of an application system to ‘take care of itself’ greatly enhances the value of the application system by reducing the limitations of application systems discussed in Section 2.2.

Definition 2: application system automated management: The establishment, relinquishment, and maintenance of an application system through provision of the necessary environmental conditions through which the application system can provide its offered services. Application system automated management can be considered a layer in-between the operating system provided services and the application-provided services.

Automated management is more complex than scripting events across multiple machines or visualizing network state. It may support elements that provide these services, but it is in itself a much more general concept. An automated management system must be robust, flexible, quick, minimally interfering, scalable to hundreds of thousands of nodes, and extensible to existing and future adaptation services.

3.1 Top-Down and Bottom-Up Automated Management

Past research suggests two paradigms for automated management of distributed application systems: top-down and bottom-up management. The paradigm of top-down control [2] [3] [4] [6] [7] [8] [9] [10] [11] specifies and imposes a design on a collection of components, and describes the adaptive rearrangements of the design. The two main categories of top-down management are descriptive specification-driven and control specification-driven [10]. Researchers have also investigated systems for automated adaptivity that involve bottom-up management. The two
main approaches in this category are ad hoc connectivity systems, such as Jini [5], and predicted emergent properties in distributed algorithms [15].

Top-down and bottom-up management approaches have distinct advantages and disadvantages. Top-down management is predictable in its effects as they are stated imperatively and acted upon. However, the complexity of specifications for large systems makes top-down specifications unwieldy or infeasible. Bottom-up approaches avoid system-wide specification in favor of specifying individual components, while proving or predicting the behavior of interaction between them. However, ad hoc service systems cannot provide explicit design structure, or require special middle-ware algorithms and specification parameters in an attempt to provide predefined structural organization [18]. Emergent property-based systems, where important properties cannot be proven to hold, require simulation to determine their complete behavior [16].

Our paradigm attempts to bridge the gap between the advantages of the two paradigms. **We employ both top-down and bottom-up management.** Our paradigm supports **specified control with emergent application of control.** The paradigm is best introduced by summarizing the two key technologies:

- **Cybernetic Nervous System of Control-Loops**
  Management is driven by top-down control of system behavior. The paradigm supports a ‘net’ of asynchronous and parallel control loops. This is often described, by morphological and functional analogy, as a **nervous system.** Conflict and stability resolution is handled by a combination of the paradigm’s framework and compiled formal specifications.

- **Emergent Communication Binding**
  Control loops are connectionless. Messages from components are multi-cast to desired receiver components. Messages are sent without specifying the location or quantity of recipients. Instead, messages specify properties that must be true at receivers. All receivers meeting these property requirements will receive the message. This means that the structure of control loops, and the placement of actuations (that describe the organization and management of the system) are run-time defined based on system state.

Specified control with emergent application allows designers and managers to specify properties of control that must be obeyed by a system. Meanwhile, it is the property state model, at run-time, that determines where control is applied in the system.
3.2 A Nervous System of Control Loops

Our paradigm applies control-driven, rather than descriptive, technology. This has three purposes:

- Rapid and concrete collection and distribution of distributed system state,
- the availability of collective decision-making algorithms for system control [11], and
- the natural control-loop paradigm for resulting component organization [22].

Control-driven management constructs programs (or imperative specifications) describing actions to be taken on a current system in response to undesirable system states. Potential system state is the input domain of a control-driven management function. Its output domain is the set of actions that can be taken. A control-driven approach, as illustrated in Figure 3 in its most abstract form, is iterative. The principal elements are a system and its controller. Sensed system state can trigger controller action. Control decisions result in actuation of commands. Actuation changes system state resulting in further sensing, and potentially, further control decisions. This iterative structure forms a control loop. Control loops have a distinct advantage. They allow incremental approaches to problem solving, altering previous outcomes from previous iterations with improvements or compensations. The iterative structure of a control system is often made more effective when a control loop is represented explicitly in control system design. We view the control loop as a useful structure for control-component organization in our paradigm.

3.2.1 Hierarchical Control Loops

It is important to process information at the right level of abstraction when making decisions. This holds true for a system controller in a control loop. Some analyses can be performed on low-level state relatively quickly. However, decisions that involve thousands of computers cannot be made quickly if the information presented to the controller...
consists of detailed low-level state from throughout the network. Therefore, controllers need to receive only relevant information at the appropriate level of abstraction in order to make meaningful decisions by reasonable deadlines. Furthermore, a single controller cannot make the right decision, quickly, for all levels of system abstraction. Added to this, a single controller would centralize an otherwise distributed system.

It makes sense to have more than one control loop, with local control loops making local decisions and ‘larger’ control loops making ‘larger’, more abstract, decisions [13]. This strategy is depicted in Figure 4. Three levels of a control-loop hierarchy are represented.

### 3.2.2 Parallel Control Loops

Multiple control loops can contribute, simultaneously, to the control of system state. Decision making about complex systems, such as application systems, often involves controllers that make decisions about specialized sub-domains of system state. Yet sub-domains of system state often overlap, or require use of common resources. For example, one controller might be an expert in decision making for tolerating non-local faults. Another controller
might be an expert in diagnosing and healing failure of a particular brand of application. These controllers might employ separate sensor and actuator networks. Their control loops should be able to operate in parallel in order to maximize system performance and response time. Parallel asynchronous control loops are depicted in Figure 5. It is important that a control system be capable of supporting multiple controllers that actuate overlapping system state.

3.2.3 The Nervous System Concept

Together, parallel and hierarchical composition of control loops represent a kind of ‘nervous system.’ An application system can be managed by multiple overlapping control loops. These loops are a network of information conduits, rapidly transferring sensing and actuating signals to and from important system state. Attached to these conduits are nodes of information processing, that process sensed state, make decisions, and specify action. In addition, some components are responsible for choosing between conflicting signals. As an example from biology, a command to point one’s arm, as opposed to pulling one hand away from pain, are ordered and carried out based on priority. ‘Nervous-system’ like constructs are a useful organization for control-driven adaptation and management of complex systems. In fact, their existence in the biological paradigm is a case of their effective use. Nervous system-like control loop is a principle of cybernetic design [23].

FIGURE 5. example of three parallel control loops
Definition 3: **nervous system**: The components of a control system, and their desired interconnections.

However, a complex nervous system of control loops has a complexity that is not easily specified. Specifications would tend to be larger and more complex than could be manageable by humans. In addition, mistakes in a design might lead to unintended system behavior that would remain dormant until certain network state occurred. In order to be effective, the control loops of a nervous system must be carefully connected. We refer to this task as **wiring** the nervous system. Wiring the nervous system of a large distributed application environment is expensive and critical.

Definition 4: **wiring**: The task of connecting complex communication pathways in a control nervous system.

The extent to which compositional abstraction can be applied is limited by the inherent complexity of application systems. The combination of varied local conditions of the environment, and the complex dependencies between application system components results in variation in actuation potential at the local level\(^1\)\(^\)\(^1\). This is very different from the situation with traditional Boolean circuitry, where **and** or **or** gates were useful abstractions and placement conditions were relatively homogeneous\(^2\).

Once a complex arrangement of control loops is established, it is expensive to modify that configuration. In biology, a nervous system is usually a static entity that is physically protected to avoid damage. Damage generally results in failure of control for at least part of the system. In a computer network, adequate security—physical or otherwise—is unlikely. Therefore, it is unacceptable to allow the cost of rearrangement or reestablishment of the control network to be expensive. While some descriptive techniques try to provide dynamic mechanisms for alteration, the propagation mechanisms proposed \([6]\) \([8]\) \([9]\) are inadequate for our expressiveness, flexibility, and time-bound requirements.

Recall that one of the goals of application system management is flexibility in the placement of application components. The control nervous system attached to an application system forms a static arrangement of connectivity to the application system. If the meta-system (the application and control system) is to support more than one ‘posture’ or arrangement of the application components, then a static nervous system would require that only one posture has

---

1. This can be mitigated by common interfaces in a component model (such as CORBA) though heterogeneity remains in a data form accessed through the common ‘interface description’ interface.
2. At current circuitry scales things are no longer this simple.
the optimal connections with the nervous system, or that a nervous system ‘mean’ design is utilized, so that its organization is not optimal with respect to particular component arrangements.

If we are to apply nervous system like control loop networks, we will need to have a technology that greatly reduces the cost of connecting its components both initially and during system operation. Fortunately, scalable methodologies to reduce the complexity of connectivity exists.

### 3.3 Selective Multicast to Replace ‘Wiring’ with Run-Time Binding

We introduce a method to avoid the problems associated with wiring a control nervous system. The wiring of components of the nervous system need not take place at all. We replace wiring between components with sender and receiver elements at each control loop component. Event messages are multicast so that a published message from a publisher element travels efficiently to all the specified receiver elements across a bus-like communications service. Utilizing this communications model, controllers can send relevant commands to desired actuators. Sensors can communicate system state information to relevant controllers. This is done by content-based routing. Information from one publisher element can be sent to multiple recipients without specifying the location or quantity of recipients.

This has several advantages over a wired control loop hierarchy. Location transparency allows control loops to be specified without consideration for location in a physical or logical network structure (unless this is desirable, as
discussed below.) Secondly, quantity transparency means that control and sensing event messages can be multicast. Multicasting events is particularly useful for connecting controllers to groups of actuators. The advantage of run-time binding is the dynamic nature of the connections, thereby maintaining the nervous system for any current installation configuration.

### 3.3.1 Multicast, Property-Bound Event Binding

There are many models of run-time bound event messaging. The use of an attribute model for binding is a particularly powerful contribution to our paradigm. Its benefits are better introduced after briefly describing attribute-based binding.

An attribute model can drive the binding of events to the appropriate receivers. An attribute model is a collection of variables and their associated typed ranges of values. An attribute model instance is an assignment of a value to each variable of an attribute model. Each receiver in our paradigm’s infrastructure maintains an attribute model instance. A selection expression is a Boolean expression that can be evaluated over attribute model instances. Event messages carry selection expressions. We say that a message $M$ matches a receiver $R$ iff the selection expressions of $M$ are satisfied by the attribute model instance of $R$.

This technique is very useful because an attribute model can describe the state of an application system. Each receiver in the communications model can utilize an attribute model to describe the local conditions that are relevant to its purpose. For example, the receiver for a Microsoft Word actuator might maintain an attribute model describing the state of the Microsoft Word installation, its run-time conditions, and its semantic state provided by a local user or administrator.

Mapping messages to an attribute model describing local state has several crucial implications. Foremost, components can message each other from the perspective of desired, run-time network state. This means that one component can speak to many components in the nervous system based upon the very attributes that make the receivers relevant to the communications. Consider the implications of this for communications between a controller and actuators. A controller may receive sensing information that has been compiled while percolating up through many layers of control loop hierarchy. When a controller decides to act based on this information, it might be many levels removed from the local actuators that must be contacted. An expressive attribute model describing the conditions that warrant controller action allows the controller to contact actuators through a description of the state that warrants contact. The controller communicates with actuators by sending a message with selection expressions describing the con-
ditions of the system that must change. In addition to describing system state, an attribute model can identify types of sensors, actuators and controllers. This allows elements to communicate only with other components that can understand a given message.

The second important implication arises when we allow sensors and controllers to modify the values of attribute models. Messages now are not limited to sending sensing or actuation events. They can be queries in a ‘distributed database’ that contains information about existing network state. For example, simple sensors might provide receivers with local state that an application is not responding to input. This local state is not collectively established. It might filter up to a controller, determining that the operating system on which they reside has crashed. In this way, the attribute model becomes a distributed model describing the network state—the element that drives system control in the first place.

### 3.4 Partially Specified, Partially Emergent Structure and Adaptive Behavior

Specified control loops explicitly prescribe system adaptation through imperative commands. Run-time multicast binding formally models run-time emergent instantiation of actuations. The potential power of the paradigm comes from their combination:

> **Structure and adaptive behavior of an application system is partially specified and partially emergent.**

The specified part of adaptive behavior, control specification, describes what must be done and the conditions that must exist where it is to be done. The emergent part of behavior is the instantiation of actuation at sites matching the required conditions. This is based on current system conditions portrayed in the attribute model. Consider the implications:

- Controllers specify intended actions without enumerating the actuation sites at which to apply intentions. This allows compact control intent specifications to scale to arbitrarily sized application systems.
- Intent specifications need not enumerate the structural arrangements of application system state under which an actuation is applied. Instead, they specify the run-time conditions required wherever an actuation is to be applied. This has two important effects:
  1. The specification size is compact as it is not enumerated for all anticipated system-state configurations
2. The specification can control an unanticipated network configuration that is described by the abstract intention specification. 

- The logical structure of the application system can be emergent result of applying installation and adaptation control intentions. 
- It is the co-definition of the attribute model and the control specifications bound to this model that states the boundary between deterministic and emergent properties of the application system.¹

Consider the structure of a large, distributed application system. A descriptive language utilizing component types and hierarchical composition can describe an application system’s structure. However, descriptive forms become extremely complex as application systems increase in scale, becoming untenable. Likewise, purely control-driven specification requires a long list of installation commands to construct the initial system structure.

In our paradigm, an attribute model can establish local properties through independently acting local sensors and/or distributed algorithms. This creates a distributed low-level application system model describing the environment and its contained application components. Consequently, control instructions for the installation of components can be based on queries to this state or on installation commands selecting desired state. The result is an emergent system structure derived from network conditions during the installation process.

The paradigm applied to system installation is equally effective for system adaptation. For example, a controller might be specified to send a message to uninstall an application at any component where certain local attributes are true (e.g., security violations have been detected.) This change in the location of the application could result in another local attribute changing (lack of a particular service provided by the application.) Consequently, another controller could re-install the application, at a new location, to restore the application’s services. The context of the activations overlapped during run-time operation of the system, resulting in an emergent behavior combining the two actions. This example demonstrates iterative problem solving through multiple controllers, intention level arbitration and appropriate attribute model building, and emergent organization.

The extent to which system properties (such as structure) are emergent depends on the data presented in the formal control specifications. If more initial attributes are provided to local nodes from a central site and detailed actuation control is specified at the control level, then the structure of the network can be largely pre-determined. On the other hand, less centrally assigned attributes will result in a less centrally-designed structure. For most application

¹. There will be systems where we want to quantify the number of installations of a component, or explicitly identify where to install it. This can be achieved by adding region and location identifiers to the attribute model.
systems, a balance of centrally distributed attributes and local property assignments will produce a reasonable, flexible, partially designed, partially emergent, actuation structure.

3.5 Intention and Resource Conflict Resolution

Previously, we discussed allowing multiple concurrent control loops to operate independently within a control nervous system structure. This will not create an effective paradigm if independent control loops can interfere with each other, preventing necessary management work from being accomplished. Within the control loop framework, there are two points where interference will hamper independent control loops:

1. **Intention Level.** The intentions of two controllers might conflict or overlap. For example, two controllers might try to solve the same problem, utilizing completely different physical resources to actuate a solution. Their attempted solutions might interfere with each other.

2. **Resource Level.** Two control loops may be attempting to actuate changes to the application system. These changes may require use of common resources in the system. Hence, the actuations may interfere with one another. For example, one actuation might try to update a file that is being written to by another actuation task.

Our paradigm supports independent control loops by resolving intention and resource level conflicts. This is handled through the provision of an intermediate service layer, called the intention layer, between controllers and actuators of each control loop. This intention layer appears in Figure 1 on page 4.

The intention layer requires that all actions to be taken by all control loops first pass through it. This layer requires that all actuations to be performed are first stated as formal intentions for desired changes to the network state. The intentions also carry the necessary information to contact the actuators that subsequently carry out the desired tasks. This layer will provide for intention and actuation level conflict resolution.

4 Architecture

We present ANDREA, the Application Network Director for Regulated Emergent Actuation, an architectural framework designed to support scalable, automated, managed adaptability for distributed applications systems. ANDREA implements the paradigm of top-down specified control with bottom-up dynamically bound actuation sites introduced in Section 3. To implement the paradigm, we will introduce two architectural level research concepts:
• **Intention Council**: Independent, arbitrary, control loops are coordinated through a resource and intention mediated workflow model.

• **Multicast Property-Bound Command**: Intended actuation workflows are multicast to workflow servers at actuation sites. Receiving workflow servers are those that have local attribute state matching the conditions required by the published intention. Furthermore, local attribute state parameterizes workflow instantiation at each receiver site. This is implemented through a combination of publish-subscribe technology and Boolean expression evaluation technologies.

In addition to its own contributions, ANDREA applies two state-of-the-art research technologies:

• **Content driven publish and subscribe event notification**: The paradigm is an effective and efficient method for one-to-many multicast event notification [20].

• **Distributed transactional workflow infrastructure**: Distributed, ad hoc, transactional workflows are state-of-the-art research technology that provide a scalable, dynamic form of correlated distributed process. They are an effective means of establishing survivability within distributed application systems [1], while providing a meaningful formal description for temporal relations between adaptation tasks [4].

The proposed architecture is depicted in Figure 7. This figure describes the control-loop framework of the architecture.

### 4.1 Requirements

There are several basic requirements for adaptation to succeed in offering a beneficial control and actuation architecture:

1. An abstracted design of the application system must be sufficient to automate its deployment, as a direct specification of the design will be unmanageable.

2. The time scale of actuations must be bound, within an order of magnitude to provide survivability adaptation under real-time failure time limits.

3. Communication bandwidth for control and actuation must scale with network size to avoid dominating network bandwidth usage.
4. We must operate assuming that as networks increase in size, we will experience asymptotically constant failure rates under normal operating conditions. When networks grow in size, it is increasingly likely that a proportion of application objects and their actuators will be non-functional at any given time.

5. Actuation of network constraints must take place in a coordinated manner. Constraints in the actuation space of application networks require conditionally effected, temporal constraints between component actuations.

6. Distributed systems have many trust boundaries requiring provision of security measures. In particular, multicast communication introduces important potential security vulnerabilities. Our system must account for trust boundaries with reasonable security measures, particularly with respect to communication interfaces.

If our architecture can meet these requirements, then it is more likely that it will find applicability in real-world application system management and configuration control. If we fail to consider these requirements, important areas of control, such as adaptability for security and survivability [14], will not be adequately supported.

### 4.2 Intention Council: Independent Control Loops

We allow an arbitrary number of control loops, as indicated in Figure 7. The capability to maintain independent control loops requires that after solution synthesis, and prior to actuation, a method to resolve control conflicts
appear. We provide this mediation by modeling the intended work of all control loops in a common intention framework. Figure 8 illustrates the introduction of this new intermediate step to existing application control technologies. The dashed arrow represents the original event flow of the control-system. Two intermediate tasks not interface the event flow. The common intention framework consists of formally specified workflows, providing a taxonomic classification of work types in the form of workflow task classes. Common work intentions, classified in the task hierarchy, instantiate to intentions that are checked for conflicts of interest based on a simple formally specified, rule-driven, finite-state machines.

An example of intention checking is cancellation of one action that is superseded by the effectiveness of another. For example, an upgrade of a software product at site set b from version 3.0 to version 4.0 and another upgrade from version 3.0 to version 3.2, at the same sites, with a plug-in component, might call for cancelling the second action. If this action is still relevant after the upgrade to version 4.0, it can be resubmitted by the synthesizing controller. As viewed in Figure 7, intention modeling occurs in the abstract actuation command, prior to multi-cast instantiation. Other intention rules would stop a conflicting action, such as actuating an application component that is in the process of being uninstalled.

The second aspect of independent control is resource contention resolution. This is handled as an extension of the classical workflow model with priority driven resource management from the operating systems literature [19]. We
will utilize a total ordered priority scheme of workflow task class intention types, with tie-breaking by instantiation
date or proposed deadline.

4.3 Multicast Selective Command Implementation

ANDREA is designed to support dynamic actuation binding as proposed in Section 3.3. This is implemented by
the combination of a content driven publish-subscribe architecture with distributed ad hoc workflow management.
We can produce dynamic binding with published workflows bound to subscribing actuation sites.

Let $\text{actSites}$ be the set of actuation sites in our system. Let $\text{attribute}$ be an arbitrary modeled attribute in our sys-
tem and let $\text{value}$ be an arbitrary legal attribute value. A function mapping from an actuation intention’s required
attribute expression to a set of actuation sites can be defined by function $C$, as follows:

\[
D_C = \{f|f(\{n|n \in (\text{attribute}, \text{value})\})\}
\]

\[
R_C = \{n|n \in \text{actSites}\}^2
\]

\[
C(f) = \{n|(n \in \text{actSites}) \land (f = \text{true})\}
\]

(EQ 1)

directly selecting a set of actuation sites as output from a set of Boolean functions over attributes and their potential
values. The function from the domain accompanies an intention and binds the intention to any actuation site for
which the carried function evaluates to $\text{true}$. This implements multicast selective command.

We propose to explore the use of publish-subscribe to achieve this function with efficiency. In doing so, the para-
digm may be limited to selection functions that are broadcast to receiver nodes prior to their use by a message.

Event driven publish-subscribe paradigms based on attribute matching [20] map workflow task attributes to actu-
ation sites, implementing function $C$ of Equation 1. Each actuation site will maintain a list of properties and maintain
subscriptions to any workflow task submissions that will match those properties. Publish-subscribe makes certain that
only actuators relevant (via function $C$) will receive an event for the potential workflow. The message may then carry
an additional selection function to resolve any false positive recipients (for bindings more specific than any previ-
ously defined channel of specification function.)
4.4 Security by Domain Access

Our workflow instantiation is multicasted from publishing control loops to subscribing actuators. A combination of authentication technology with partitioning of a system into domains allows access control over actuation. Actuators can restrict the domains from which they accept workflow submissions. The domain model can be built into the attribute model, so that attribute binding can only occur for messages from authorized publishers to authorized receivers.

5 Preliminary Research Results

Our preliminary research has demonstrated the use of independent control within a common intention framework. To implement separated control, we introduced distributed resource contention resolution and intention-based prioritizing (a primitive form of intention arbitration.)

The architecture for this preliminary design, known as Willow [14], is depicted in Figure 9. The reactive control loops, RAPTOR and Administrative Workbench, independently control actuations. These actuations were mediated by ANDREA and actuated by Software Dock. The following describes the role of each component:
• *Software Dock:* A universal application installation and management system driven by a property-based rule base defined in XML. Software Dock maintains a model for local application state. This serves as a universal software management actuator [21].

• *Siena:* a content-routing publish/subscribe event notification system. This serves as our publish-subscribe communication layer for attribute driven communication [20].

• *RAPTOR:* a rapid non-local fault tolerance system. It is driven by compiled fault tolerance specifications. This serves as an autonomous control loop component [11].

• *Administrative Workbench:* A console-driven system administrator’s network application system adaptation workbench. This serves as another independent autonomous control loop component [21].

• *ANDREA:* Adaptation Needs Director with Reactive Emergent Actuation is a network-state, workflow based, common actuation intention model, with resource contention resolution. It provides arbitration of actions from independent control loops.

The author’s contributions to the experiment included the conceptual integration of the Administrative Workbench and RAPTOR systems, together with the prototype ANDREA model for workflow-based intention arbitration and resource-conflict resolution. The prototype of ANDREA consists of several subcomponents, each of which can run anywhere, but are likely to be distributed at all actuation sites as part of the actuation framework. ANDREA’s subcomponents, present in Figure 10, are as follows:

• *Resource Management:* a distributed system for hierarchical resource management. Resources are obtained by workflow task instances in order to allow them to execute. Completed, failed, or aborted workflow tasks release held resources. Resource hierarchies are specified by XML-like input to the resource management system.

• *Workflow Management:* a distributed system for hierarchical priority and dynamic resource driven workflow actuation. Workflow managers maintain two separate data states. The first is an XML-like specification of workflow task types. Each task type maintains attributes pertaining to its low-level actuation commands, as well as a statically assigned priority. The second data-state is the set of workflow tasks instantiated in the current workflow system. Each workflow task is driven by a finite-state machine. The states of the finite state machine include waiting for other tasks and waiting to collect required resources. The tasks support preemptive earliest deadline first scheduling. A linear ordering of priorities of task types (with tie breaking by timestamps) prevents deadlocking in complex active workflow graphs.
• **Workflow Synthesis**: a simple system for specification and storage of pre-defined workflows. This component specifies workflows in an XML-like notation.

We investigated the automated actuation potential of our system through construction of a small scale distributed application system. This system was designed to mimic some basic features of the Air Force’s proposed Joint Battlespace Infosphere, referred to as the JBI. This application system consists of a virtual communications network and a set of applications communicating through this virtual network. The schematic for this system is presented in Figure 10.
11. The large rectangles are computing nodes. In this case there were five Windows 2000 machines. The shaded boxes and the circles represent application components of the “mini-JBI”. Arrows indicate communication paths between the application components.

5.1 Experimental Results

Willow provided automation of the following tasks via the two control loops:

1. The RAPTOR Control Loop
   - Reactive actuation resulting from a JBI communication link failure
   - Reactive actuation of security protocols for running JBI components

2. The Administrative Workbench Control Loop
   - Automated installation, activation, deactivation, and deinstallation of the entire JBI application system
   - Installation, activation, deactivation, and deinstallation of individual application components

The demonstration system verified that decentralized control loops can cooperate within a common actuation intention framework. A primitive form of Intention Council was an effective means of combining proactive and reactive adaptation controllers. ANDREA demonstrated that its workflow-driven model was sufficient for intention and resource mediation between the two control domains on a small distributed application system. Asynchronous control loops worked efficiently under testing conditions. This indicates that the basic design of Figure 5 on page 12, is feasible in the small scale.

In addition, we demonstrated that a majority of administrative overhead can be stated in formal specification, thus providing scalability for task complexity in small distributed systems. For example, during testing of our “mini-JBI,” we were able to run repeatedly and install 11 application components (in a required order without human intervention) through a formal specification. Total installation and run setup actuation time took less than 2 minutes. This greatly reduced administrator workload. It also decreased time requirements to place and organize the test application system.
6 Proposed Research and Evaluation

We propose research to investigate several facets of the introduced paradigm: characterization, development, and effectiveness. We are interested in continuing to characterize the paradigm in order to determine its strengths and weaknesses. At the same time, we wish to develop the paradigm to a level of implementation, so as to better understand the design space available for its potential application. Once developed, we can determine the paradigm’s effectiveness for adaptability and maintenance of large application systems.

6.1 Characterization

The paradigm’s algorithmic potential is not fully understood. Characterization of the paradigm requires that we better understand several aspects:

- its computational and programming model,
- its performance, and
- its service capabilities for a system.

Each of these helps identify the implemented paradigm’s range of application.

We must determine the kinds of emergent properties that can be result from emergently placed actuation, as well as those that can be achieved by parallel asynchronous control loops. Our paradigm initiates and modifies emergent properties via top-down control. We want to determine the extent to which succinct control specifications can describe control initiating and modifying potentially complex emergent adaptation. We also want to determine the extent to which meaningfully succinct control specifications can counter or control emergent properties resulting from actuations.

We are interested in the predictability of pin-pointed system properties. Particularly, we wish to determine the predictability of measurable system properties that inform human administrators of system state. The effect of specification of controllers, the chosen system state attribute model, and run-time system conditions on this predictability must be characterized to guide useful application of the paradigm.

The stability of the paradigm may depend on the specifications and attribute models applied to the system. We must determine how an intention council must be programmed (via specification) in order to maintain desired system state. An overall characterization of programming in this paradigm should be determined. A comparison with existing programming paradigm’s will be considered for classification of the resulting programming exercise. Furthermore,
we plan to analyze the programming task, and the applied algorithms, to better determine its place amongst computing models. Its relation as an instance of the cybernetic model is the most interesting as of this writing.

6.2 Development

We have several algorithms to perform paradigm tasks with special technology. In particular, a form of multicast property bound messaging using publish-subscribe and Boolean satisfaction has been discussed. In addition, intention council as priority level system with transactional semantics over workflows and a finite state-machine rule-base has been briefly presented. However, we do not currently know if those are the most efficient or most appropriate algorithms and technologies for the paradigm’s features. In addition, there are likely to be other design considerations that we have not yet taken into account.

We need to enhance and refine algorithms for the paradigm. We require algorithms that are sufficiently expressive to support adaptability and management for application systems. In addition, the algorithms are further constrained in that they must be efficient enough to support fast actuation of complex tasks in, if possible, bounded time. Furthermore, the algorithms must scale well for very large systems. Distributed algorithms will be favored for their fault-tolerance characteristics. Algorithms will be considered for their potential to enforce meaningful security policies. We will examine intention council with these considerations. Multi-cast attribute-driven message binding (leading to multicast selective command) will also develop with this approach.

Development will depend on research into effective technologies with which to implement the paradigm. Currently, distributed ad hoc workflow technology and content-driven event publish subscribe systems appear to be key technologies for implementation of the paradigm. Determination of algorithms that would use other mechanisms might shift this focus. One particularly important implementation issue remains: multi-cast selective command implemented on a workflow engine and publish subscribe system.

6.3 Effectiveness and Utility

How useful is the paradigm for maintenance and adaptability in large scale distributed application systems? This question will largely determine the utility of the paradigm for maintenance of application networks. Another concern will be vulnerabilities that the paradigm might add to traditional application systems. We will attempt to identify these vulnerabilities where they cannot be eliminated by paradigm implementation choices. We can also determine
the relative importance of the main features, algorithms, and technologies in the successful or failed application of the
paradigm. Effectiveness will be further explored by measuring viability [23] as parameterized by system scale and
desired system adaptability. The utility of the paradigm can be determined by qualitatively comparing its effective-
ness with its costs.

6.4 Evaluation Methodology

Development will play a fundamental role in our evaluation process. Throughout the research, we will be devel-
oping our implementation of the paradigm. Development will proceed in stages, and therefore, evaluation will also
occur in stages. Our goal is to conduct characterization and effectiveness research in parallel with development whenever this is possible. However, some characterization and utility research requires development milestones to be
achieved before the apparatus, either analytic or evidential, exists to test relevant hypotheses. In an ideal research environment, we could explore the entire design ‘space’ of development possibilities. However, we will be limited to
making design decisions based on the results of characterization and effectiveness measures during the development process itself. Hence, there will be an inherent iteration in our ‘effectiveness’ evaluation process. (The ultimate assessment of characterization and effectiveness will depend on our chosen design path—one that will have resulted from earlier effectiveness evaluations.) This iteration dependency will be unavoidable given our limited resources.

In attempt to compensate, we will stress two goals:

• careful documentation and presentation of our development design decisions, so as to present arguments for our
  chosen path from the design space, and

• identification of crucial points during the design process. These design points will yield reasonably comprehe-
sive results and will facilitate making decisions during evaluation.

6.5 Process

The stages of experimentation are depicted in the graph of Figure 12. Our research is represented as a process
with experiments, proofs, analysis, inputs, and resulting evaluation products. Analyses appear in the graph as circles.
During the course of our research, there are five main analyses that we will perform:

1. algorithm proofs: proof of the theoretical performance of the developed algorithms.
2. *algorithm simulations*: simulations designed to evaluate algorithm performance and sufficiency for our paradigm’s requirements.

3. *programming and computational paradigm analysis*: determination of the paradigm’s resulting programming and computing models, and comparison with existing programming and computing models, to determine the expressiveness and computational power of the paradigm.

4. *domain programming experiments*: application of the paradigm’s programming model to meet management and adaptation specifications from prepared application system domains, resulting in an evaluation of the adaptive system capabilities. This provides both characterization and effectiveness information.

5. *systems model evaluation*: analysis of previous results, including domain experiments, with respect to defined architecture models of adaptive software systems, such as survivable software [13], intrusion tolerant systems, ubiquitous systems, and viable systems [23].

Rectangles represent input and output from evaluation analyses. The evaluation products of our research are:

1. *implementation components*: components designed and implemented to execute the paradigm. These will be developed from our preliminary algorithms from our previous research and our algorithms sketched in this proposal. In addition, they will be influenced by feedback from algorithm proofs and simulations that we create to evaluate the components.

2. *performance characterization*: proofs of algorithmic performance and capabilities, along with statistical evidence collected from simulations. Examples of capabilities include fault tolerance, security, communications efficiency, system size scalability, programming scale, and bounded response time. These results will come from analysis, experiments with implemented components, and simulations.

3. *programming characterization*: the categorization of the computational model, and programming model (input to the computational model) of the paradigm. Experiments with the implementation and simulation can verify results produced from analysis for attempted categorizations.

4. *effective property characterization*: a characterization of the types of controlled properties and emergent properties that can be effectively created and enforced by our paradigm. Implementation and simulation experiments (based on implementation architecture) will provide data for analysis. In addition, domain programming experi-
ments will quantify expected and unexpected properties that occur or do not occur in implementation experiments.

5. adaptation and maintenance effectiveness/utility: conclusions drawn about the extent to which behavior of an application system provided with the services of our paradigm met the required properties for application system models considered. System models, including survivable systems, viable systems, and intrusion tolerant systems, describe behavior requirements for adaptive systems. Behavior data for application domains can be analyzed to determine effectiveness in the synthesis of an application system meeting the requirements of the given system models. Comparison with costs of using the paradigm qualitatively characterize the paradigm’s utility.

Early evaluation focuses on development, while later evaluation focuses on effectiveness assessment, with characterization being the primary intermediate phase. After each development cycle stage we will refine the characterization of the paradigm.

After the development phase, the experiments shift to characterization of the programming and computational models of the paradigm and experiments with the paradigm for specific application systems. These elements deter-
mine the character of programming methods, expressiveness, and power of the paradigm. Experimentation at this stage will consist of use of the paradigm implementation as well as additional simulations, to extend experiments to scales for which resources are not available.

### 6.6 Measurement

Measurement frameworks in which to conduct experiments and simulations can be parameterized by several factors:

- system size (system scale),
- external event scenarios (event scale),
- adaptation and maintenance specifications (constraints/requirements scale), and
- programming complexity (programming efficiency scale.)

From these parameterizing factors, we will obtain direct measurement of the low-level state of application systems over time, measured as:

- resource usage,
- correct application execution,
- maintained application presence, and
- correct, meaningful, and environmentally responsive application configuration (relative to environment conditions).

To the extent possible, all of these factors will be evaluated by measurement of operational implementations. Where the range of these factors extends beyond our ability to test our implementation, we will utilize simulation.

Properties describing the structure and behavior of the application system will also be of importance. These properties can be composed from direct measurements of application system state (as discussed above), either over location, time, or both. From such low-level properties, higher-level properties are composed. For example, low-level predictability and emergence properties may contribute to overall properties such as security and intrusion tolerance that in turn may contribute to survivability and viability. Therefore, we should be able to measure for the presence of properties such as emergence and predictability from simulations and implementation experiments, and from these, determine the presence of higher level properties.
Effectiveness, and characterization of the paradigm can be measured by comparison of pre-programming specifications (application domain specifications) with observations of implemented and simulated application system behavior. Identifying and cataloging the properties from pre-specifications that successfully appear in the experiments measures the degree of success in programming the application domain in the paradigm. It will also determine which properties are most easily expressed in the paradigm. It will not exhaustively determine applicable properties, or environmental situations under which demonstrated properties will still hold. By creating pre-programming specifications to contain the properties required for system models such as maintainability, survivability and viability, we will be able to address the applicability of the paradigm to the models for which automated adaptation was intended, and hence, the effectiveness of the paradigm. By comparison of effectiveness with the work required to obtain the effect (scale of specification and time investment), we can determine the paradigm’s utility.

7 Related Work

7.1 Software Architecture Description Languages

Software Architecture and Configuration Languages provide a means for the formal specification of component-based software design. Several software architectural description languages (SADLs) and configuration languages (CLs) have been tailored for large distributed software systems.

Darwin is a SADL supporting hierarchical component-based composition with service-driven interconnections. It recognizes the need to separate component physical organization from compositional service structure [8]. Darwin supports adaptation through dynamic component propagation specified within component services. This is a limited form of actuation for system service mobility that is expressive but potential unscalable. PCL is a CL for modeling evolving system architectures. PCL attempts to model the configuration space of system by explicit description of its configuration space [9] with the extension of parameter driven ad hoc polymorphism. PCL attempts to model the configuration space in there dimensions of software and its environment [9]. ARC [6] is a compositional software architecture language that assumes an underlying service binding architecture, such as Jini. Arc relies on dynamic instantiation of components with automatic ad hoc service binding as its means to establish dynamic network state. Darwin, PCL, and ARC largely rely on hierarchical compositional structuring and are data driven, as opposed to the control-driven approach of our paradigm. In addition, Darwin and ARC are concerned with data-driven service struc-
tures. PCL conducts configuration management through explicitly parameterized descriptions of configuration. PCL and ANDREA share in common that they used attributes to parameterize actuation.

Manifold [10] is a CL that, like ANDREA, is control driven. It utilizes controlling managers that are responsible for care-taking the communications behavior of application components. These controllers can help specify real-time requirements and coordination of between primitive components. Hence, Manifold separates control from actuation, and utilizes control to derive configurations. A workflow method exists for performing operations similar to Manifold. This system utilizes workflows to describe data-requirements between independent components. Much like I/O managers describe the event coordination in Manifold between otherwise dumb components, Workflow task instances coordinate activities between simple components [3]. Manifold and the workflow system require explicit specification of actuation sites in parameterization of controller/application instantiation. This explicit linking of components will not scale well to very large systems, where the number of components is very large and not directly measurable, and references to individual components by name or locality are unfeasible. A potential solution to the design linking problem is to rely on a dynamic service binding system such as Darwin or ARC. This still requires that component instances be actuated by location, however, and this will not scale well.

In addition, they do not support the introduction of control loops other than those directly specified by their control languages. Therefore, there control technology is only extensible if other’s control systems are rewritten in specification form. Analytic control programs cannot be utilized. Nor can control loops be active independent of one another as with ANDREA.

7.2 Emergent Property Adaptation

Emergent properties from local actuation can be a means to establish total application system adaptation. This is a bottom-up means of control for which overall behavior is emergent [15]. This method does not require control-loop driven actuation. However, the resulting systems require simulation in order to determine their collective behavior, as it is not specified. This unpredictability might hinder the paradigm’s usefulness if engineering cannot make its effects desirable. In addition, emergent adaptation requires action on high level abstract threats, not present in observable local system state, via local, low-level components. This may not be feasible. While adaptation to properties that appear at low levels is appropriately handled in a distributed algorithm with an emergent result, many high level faults and threats will not be effectively handled in this manner, particularly in a bounded time.
### 7.3 Dynamic ad hoc (unspecified) Component Networks

Several architectural standards exist for the creation of dynamic ad hoc networks. Jini [5], Corba, and DCOM provide service discovery standards. These standards utilize service lookup to create ad hoc service networks. They perceive actuation as the installation and removal of service instances. This is bottom-up emergent actuation of inter-component design.

Gryphon attempts to remove the entirely application-level ad hoc location policy for application components. Gryphon recognizes that application locality (relative to other application instances) effects overall application performance. The gryphon methodology utilizes a middleware agent approach to mitigating the placement of components. This is a form of dynamic actuation binding limited to the location of application components. Gryphon is a control loop that, through auxiliary interfaces, allows components to help influence their own placement in a more expert manner. Our approach to control loop independence would allow Gryphon to avoid operating as a middleware layer. Placement of components becomes specification driven, and it is only as emergent as specification attributes allow. This is a generalization of the specific targeted aim of application efficient location that is the goal of the Gryphon technology.

### 7.4 Collaborative Agents

Collaborative agents attempt to solve problems by working together to mutually actuate a solution to a given problem. Our paradigm does not involve collaboration between controllers. Any collaboration between actuators is assumed to be well understood prior to reaching the actuators and specified in actuation intent workflows. There is no reason, however, that agent-like actuators attached to our paradigm could not work in such a manner.

### 7.5 Amorphous Computing

Amorphous computing involves the arrangement of common component types into meaningfully abstracted computing elements. It is primarily research for circuit design at scales that preclude explicit complete design specification. This use of an algorithm to avoid direct specification of all design characteristics, by specifying a process rather than a result, is similar in spirit to our approach. However, amorphous computing generally involves more specific algorithms for hardware organization. The general algorithms of amorphous computing, when applicable to application systems, might be applicable through our paradigm.
7.6 Cybernetic Control

Cybernetics is perhaps the most directly relevant parallel contribution to our research. The Cybernetic organization developed by Herring [22] represents many of the same elements as our research within a formal model. Herring's Cybernetic model is based on the Viable Systems Paradigm invented by Stafford Beer. This model recognizes separation of control from the controlled entity, and more specifically, the nervous system analogy (parallel and asynchronous control loops) [22]. Herring also recognizes the importance of the human operator as a specific layer of a control hierarchy from which automation can be derived [23], and the use of expert systems as controllers [25]. This is a form of migration from human to automated control [23]. Herring’s cybernetic model also recognizes the need for scheduling and coordination between actions [25]. The basic control theoretical notion of the requirement of a model in order to establish control is explicitly employed in his work. [22]. He also utilizes the notion of workflows and transactions to negotiate resource conflicts [23].

In essence, our work can be seen as an application and extension of the cybernetic model. We introduce the notion of location independent message binding. (This simulates both ‘molecular messaging’ and the connectivity of neurons (which establish connections over large distances through chemical signals during neural growth.) as an appropriate mechanism for binding cybernetic control components. Furthermore, we maintain the model-dependencies of control, instituting message binding utilizing data in the explicit system state model.

Herring argues that complex systems are self-similar in scale-view organization [23]. This author disagrees, arguing that much of the functionality of complex organic systems comes from their varied organization at compositional scales. This is demonstrated in the organization of the nervous system from the eye to the image processing centers of the brain. Heterogeneous organization creates the complexity of computed functions at each scale. At smaller scales, molecular signals differentiation similar-looking cells within sub-functions. Therefore, we do not feel that a pattern language will adequately represent large system’s inherent need for heterogeneity. Where it may apply, as Herring suggests, is in the self-similar use of cybernetic control loops at all scales.
8 Bibliography

8.1 References


*Working Conference on Complex and Dynamic Systems Architectures*. Brisbane, Australia. December 12-14,  


### 8.2 Sources


