

The Viable System Architecture

Charles Herring and Simon Kaplan
Department of Computer Science and Electrical Engineering
The University of Queensland
Brisbane, Queensland QLD 4072
{herring kaplan}@dstc.edu.au

Abstract

This paper presents the Viable System Architecture as a high-level reference architecture. It is component system architecture motivated by an emerging class of applications we classify as “complex systems.” The architecture is based on the Viable System Model: a cybernetic model of organizations. The concept of Viability is introduced as the overall quality desired of software for complex systems. We explain how viability is achieved by the interaction of a number of principles: autonomy and adaptation; recursion and hierarchy; and invariants and self-reference. The special structure of a component in this architecture is described in detail. The nature of an interface is also described. This unique component interface mechanism defines the component framework and provides for dynamic assembly of systems of sub-systems. We present an outline of a business-to-business e-commerce application to illustrate the qualities and principles expected from software systems developed based on the architecture. We are currently building a prototype of this system to verify and validate the architecture.

1. Introduction

We identify a class of software systems as being “complex systems” [1] [2]. Examples of complex systems include Smart Environments, Ambient Computing, Multi-Agent Systems, Adaptive/Intelligent User Interfaces and Business-to-Business e-Commerce. Building this type of system raises new software engineering challenges, and new software architectures are required to provide design and implementation guidance. Complex systems are characterized by large numbers of heterogeneous components with a high degree of interconnections, relationships and dependences. They exist in a dynamically changing environment that demands dynamically responding behavior. In other words, these systems must adapt to their environment.

We have developed the *Viable System Architecture* to address the requirements of this class of software. The architecture is based on a Cybernetic control theory model called the *Viable System Model* [3]. Our architecture defines a unique set of component interfaces that in turn defines the framework itself. We believe the special nature of the framework will permit development of protocols for dynamic assembly of systems from sub-system frameworks. We describe the architecture in detail and present an example application in the business-to-business e-commerce domain. We are currently implementing this application.

2. Beyond Objects, Beyond Components

In the paper “*Beyond objects: A software design paradigm based on process control*” [4], Shaw analyzes the “canonical” object-oriented design problem – the automobile cruise control. Her fundamental insight is that *it is a control system* and there is a body of engineering knowledge associated with such systems. She develops a software architecture, the “control paradigm”, based on classical feedback control theory (as opposed to an architecture drive by a generic software development methodology.) A diagram of the cruise control system is shown in Figure 1.

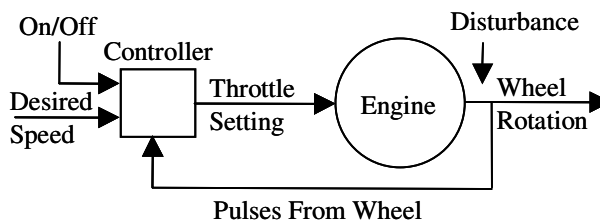


Figure 1. Control Architecture for Cruise Control

She notes the control paradigm separates the *plant* (the Engine in this case) of the main process from the compensation for external disturbances, the *control*. This separation of concern yields appropriate abstractions that

lead to design issues that might otherwise be missed in a domain neutral software development approach. In particular, performance and correctness constraints are identified early in the design process.

Shaw offers the following “rule of thumb” for use of her control paradigm architecture: *When the execution of a software system is affected by external disturbances, forces, or events that are not directly visible to, or controllable by, the software this is indication that a control paradigm should be considered for the software architecture.*

In analyzing the cruise control problem Shaw recognized that control theory was the most appropriate context for the problem. *We claim her rule of thumb is the general case for “complex” software systems.* We maintain that these types of software systems should be designed to adapt to external disturbances. Control theory in general and Cybernetics in particular offers an approach to doing this. If the underlying theory and principles of Cybernetics are in fact general, then there is no choice – they cannot be avoided in any non-trivial information processing system. For example, we have shown that the most successful design pattern, Model-View-Controller, is a closed-loop control system [5].

In “*Software Components: Beyond Object-Oriented Programming*” [6], Szyperski provides the following definitions:

- Software **components** are units of independent production, acquisition deployment and extension. (The two most well known component systems are COM and JavaBeans.)
- A **component framework** is a dedicated and focused architecture, usually around a few key mechanisms, and a fixed set of policies for mechanisms at the component level.
- A **component system architecture** consists of a set of platform decisions; a set of component frameworks; and an interoperation design for the component frameworks.

Szyperski outlines the relationships between patterns, frameworks (first-order frameworks like those based on Model-View-Controller), components and component frameworks (second-order frameworks). He describes component frameworks as subsystem architectures and believes the next step in component-based software is to develop, study and understand these second-order frameworks. The long-term goal is to realize system architectures as “systems of subsystems” or framework hierarchies.

3. The Viable System Architecture

We now describe the Viable System Architecture (VSA). As stated above, the intent is to address the requirements of complex systems. We are motivated by

Shaw’s observations. Our goal is to specify a component framework in such a manner that it may be extended to support the systems-of-subsystems concept in the form of a hierarchy of self-similar frameworks.

To this end we have chosen a model called the Viable System Model (VSM), developed by Stafford Beer, as a starting point [3]. Beer developed the VSM for the purpose of modeling and understanding human organizations (enterprises). It is based on his study and observation of many organizations over a thirty-year period. His goal was to discover their invariant structures and behaviors and describe them based on cybernetics. From a software viewpoint, the VSM can be thought of as the pattern language of complex systems. The VSA is a High Level Architecture based on the VSM.

3.1 A Quick Tour of the Viable System Model

A basic understanding of the VSM is necessary at this point. The following is a brief introduction to the features and structure of the model. A simplified diagram of a viable system is shown in Figure 2. Compare this with the standard control system shown in Figure 1. Note the fundamental separation of control and object of control or plant. A difference from Figure 1 is the internal structure of the controller and the internal structure of the plant is shown. The particular roles and their relationships are central to the VSM.

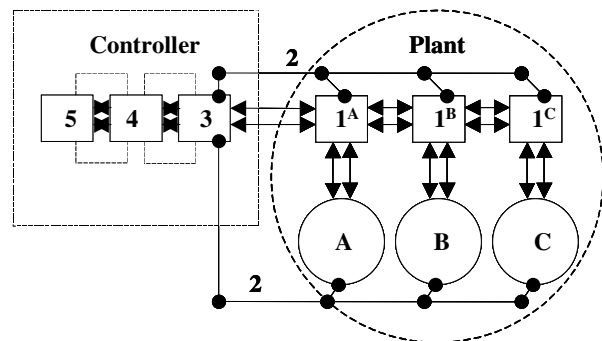


Figure 2. Simplified Viable System Model Diagram

The numbered boxes and circles have the following function or role in the viable system model:

- 5: Executive. Steering and policy; external interface.
- 4: Planning: Anticipation of change caused by external environmental disturbances.
- 3: Operations: Concerned with direct operation of the Plant.
- 2: Regulation: Production scheduling, prevention of oscillation, audit and inspection.
- 1^{A-C}: Controllers at the next level of recursion.
- A-C: The Plants.

The major aspects of the VSM are now described. First, the internal structure of the controller is related to control system terminology. A controller with just function 3 (Operations) present is a standard controller (as shown in Figure 1.) The addition of function 4 (Planning), which is coupled to function 3, is called an *adaptive controller*. That is, function 4 acts as a “tuner” or meta-controller of function 3. The addition of function 5 (Executive), which is coupled with the *4-3 couple*, is called a *supervisory controller* and acts a meta-controller for the 4-3 controller pair. The next feature to observe is the recursive nature of the VSM. Inside the 1^{Ac} controllers same 5-4-3 pattern is repeated. However, they are specialized for the particular control task at that level or recursion. Likewise, the entire system is a recursion of a higher-level viable system. In terms of control theory the VSM can be classified as a hierarchical, autonomous, intelligent, supervisory-adaptive control system [7]. Autonomy is the degree of freedom the level 1^{Ac} controllers have in terms of decision-making and adaptation. Intelligence is a matter of sophistication of the control algorithms of the 5-4-3 system.

3.2 Qualities and Principles of Viable Software Systems

Viability is the overall quality that software systems based on the VSA should have. At the highest level of abstraction viability means a system can “maintain its identity.” What does this mean in terms of software systems?

At one level the answer is a viable software system can maintain stability. Obviously this relates to the control theory approach. Adaptation is a mechanism used to achieve this. Now, complex systems live in a complex environment, which is to say they are simultaneously embedded in many other systems. So maintaining stability is a multidimensional problem requiring many interrelated strategies. From a software engineering perspective we think the notion of viability subsumes a host of “ilities”: reliability, scalability, understandability, maintainability and so on. These are facets of the viable system. How can the architecture help make viable systems at the component framework level? Our model, the VSM, says that viability is achieved by the interaction of a number of principles: autonomy and adaptation; recursion and hierarchy; and invariants and self-reference.

3.2.1 Autonomy and Adaptation. Autonomy relates to the degree of freedom for local decision-making. It is always listed as an attribute of Agent-like architectures. In our architecture it is essential for flexibility in the component frameworks. It is also essential for distributed systems. Autonomy is specified by higher subsystem frameworks and bounds the nature and degree of adaptive

behavior. For example, autonomy can be specified via policy that grants permissions for certain types of activities.

Adaptation is a key principle and we classify the adaptive capabilities of a viable software system as follows: homeostatic, morphostatic and morphogenetic. These correspond to the 3, 4-3 and 5-4-3 functions of the last section. Homeostasis is the maintenance of critical variables within certain limits to ensure stability of a system in response to changes in the environment. This is normal control system behavior. Morphostatic and morphogenetic adaptation relate to *structural adaptation*. Morphostatic behavior is “simple adaptation” such as changing internal control algorithms (e.g. adaptive control). This corresponds to the 4-3 function. Morphogenetic systems maintain meta-properties of the system (“identity”) through evolution of the structure and/or components that make up the system itself. That is, the ability to acquire new components and discard others. This is the full 5-4-3 controller function. In short, a “supervisory-adaptive” controller [7] for a system implements homeostatic, morphostatic and morphogenetic policies to manage (maintain the stability of) the plant. The viability of a system is a measure of how well these policies are realized in a particular environment.

3.2.2 Recursion and Hierarchy. In the brief introduction to VSM of the last section we pointed out that the model is inherently recursive. Each subsystem layer contains all the layers beneath it. Likewise, each subsystem it is also contained by the levels above it. The VSM is a hierarchical control system. Recursion and hierarchy are strategies used by systems to manage complexity. Functionality must be partitioned off to lower levels otherwise the system is overcome by detail and management is impossible - decision-making cannot scale. Each level performs a unique set of functions. Although there are similar patterns at all levels, they are not equivalent. Recursion and hierarchy also relate to autonomy discussed above. These principles are central to the goal of our architecture, namely the development of systems based on subsystems or framework hierarchies. They are also related to the principles described next.

3.2.3 Invariants and Self-reference. There are certain key structural and behavioral invariants in the VSM that are carried over into the VSA. The fundamental separation of control and object of control (plant) is one. The internal organization of the controller is another; the 5-4-3-2-1 functions described in the last section. The requirement for autonomy at each level is yet another. Recursion and hierarchy are also invariants in the viable system. These invariants give rise to the self-similar (fractal-like) structure of the system. Once these are understood, any sub-system level can be understood.

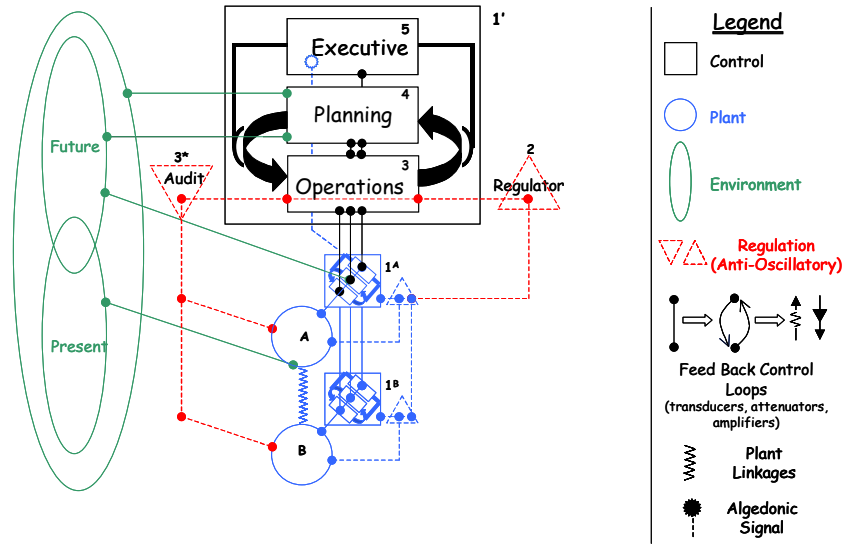


Figure 3. Viable System Model Diagram

To a large degree the concept of self-reference embraces many of the desired principles of the architecture. Self-reference is the property of a system such that each part makes sense in terms of the other parts. The system defines or produces itself based on the parts and their arrangement. This property is also called logical closure and is related to identity, self-awareness, self-repair and recursion itself.

The presentation so far may seem very abstract, but we claim it is the theoretical underpinning necessary to achieve the type of software component framework required to support the design of complex systems. The task of the next section of is to make these abstractions concrete.

3.3 The Viable Component Framework

The preceding sections have provided an overview of the VSM and presented the qualities and principles of the VSA. Here we go into more detail on the structure of the architectural model. The goal is to arrive at a high-level component description.

Figure 3 is a more accurate representation of the structure of the VSM. Note that it is basically Figure 2 rotated 90° and with some added detail. The main controller functions 3-4-5 are the same as described above. The ellipses emphasize the external environment in which the system is embedded. There are two sub-systems in the diagram. The circles in the center of the picture (A and B) represent two plants - software components that do the work of the system – and their controllers (1^A and 1^B). Also note that in this diagram the linkage between plants is shown. That is, the output of one plant can be the input of another. The rectangle (1') represents the controller.

The controller is connected to the plants, which it monitors, and performs the executive, planning and operations (5-4-3) functions.

We expand on the *anti-oscillatory* circuit that is shown in dotted lines: **Regulator (2)** and **Audit (3*)**. These functions provide needed feedback loops to ensure smooth operation of the overall system. The Regulator is responsible for maintenance of the production schedules and for coordination with other controllers. (These schedules or plans come from the controller via operations.) The Audit function monitors the behavior of the plant and passes that information to operations for processing. This can take the form of random or sporadic inspections of the plant. This is in addition to the normal or routine reporting that goes on between operations and the plant. It is an additional error detection mechanism. Finally, an “algedonic” (pleasure and pain) signal is shown going from the controllers at the lower level directly to the executive function at the next level. This can be thought of as a “panic override” alert that bypasses all filtering in the 3-4 functions.

3.3.1 Viable Component Interfaces. We are now in position to determine the VSA’s component interface specification. We proceed as follows. On Figure 3 we draw a closed curve around one of the components, 1^A+A, for example. If we now examine each of the lines that cross that curve we will have identified the set of interfaces of a viable component. Figure 4 shows the result. Thus, a viable component has, at most, nine types of interfaces. This set of interfaces is the “contract” between producer and consumer or client and server in the framework. A brief description of the generic interfaces follows:

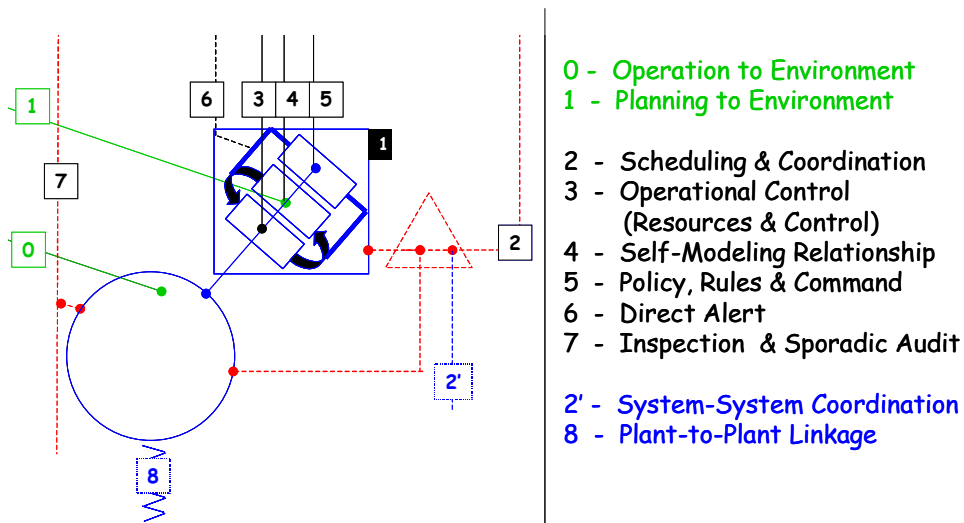


Figure 4. Viable Component Interfaces

0. Direct Plant to Environmental coupling. Provides environmental information directly to the Plant and the Plant provides information to the Environment.
1. Planning's "view of the future" environment. This may be from historical data, data projections or simulations based on the current environment.
2. Provides for transmission of plans and schedules from the upper level controller's scheduling system to the component. Also for coordination at the same level of recursion (2').
3. Commands and resources flow from the upper level controller's Operations function to the component. Routine status reporting and requests for resources flows from component to upper level controller.
4. This establishes the "modeling relationship" between components. The component transmits a model of itself for inclusion in the upper level controller's self-model for use by its Planning function.
5. Upper level controller policies and rules are transmitted to the component's Executive via this interface.
6. This interface is used by the component to directly signal the upper level controller's Executive (bypassing Operations and Planning) when needed, e.g. panic.
7. The sporadic audit interface permits the upper level controller's Operations to monitor and verify the component's state.
8. This is the direct Plant-to-Plant connection. Plants may be chained together in production lines.

This set of nine interface types is the key construct of the architecture. They are the mechanism by which the principles described earlier are achieved. *Given the recursive and self-referential nature of the model; the viable component interface specification defines the framework itself.* This arrangement provided for a self-similar system of sub-systems component architecture where any level may be treated as a component or a component framework. From an interface perspective they are equivalent.

3.3.2 Interface Structure. An interface defines the input and output specification between two system elements. Some examples are the interfaces between Operations at one level and Operations at the next level down, between the Plant and its Environment and between two Plants. In the VSA all interfaces are feedback loops (see the legend in Figure 3). These interfaces have a particular structure that is shown in Figure 5.

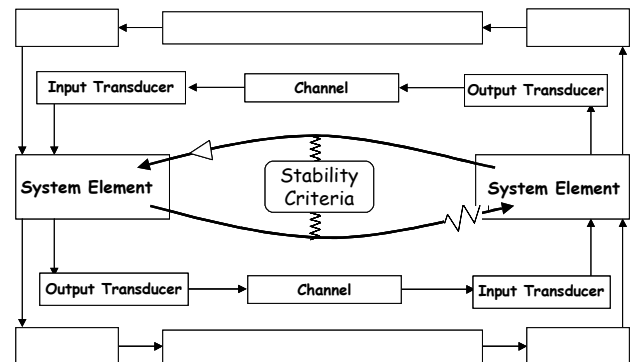


Figure 5. Interface Structure

Interface design begins by establishing “stability criteria” for the interface between the two systems. For example, if a component is transmitting at 2.4 GHz, the receiver must be capable of receiving at that rate. Otherwise, the receiver’s internal buffers may overflow, causing retransmissions, etc. In other words, the system will be driven out of equilibrium and may fail thereby affecting other parts of the system. Based on the design criteria for the interface, input and output “agreements” are designed and implemented. The general form of such an agreement is a channel with transducers at each end. The channel is the means for communication. Transducers may be needed to match input/output impedance. For example, information is encoded in some format (XML), transmitted (HTTP) and decoded into another format (XSL to HTML). There may be multiple agreements involved in one interface. The design criteria must be developed to accommodate them all.

The extent to which the formation of these interfaces can be made dynamic is critical if automated assembly and disassembly of component ensembles is to be achieved. Depending on the application domain, existing technologies appropriate to the application such as Jini [8], Universal Plug and Play [9], or workflow protocols such as Orchestration [10] can be used. This is the subject of the next section.

3.3 Component Transfer Protocol.

The last major feature of the VSA framework is support for dynamic component assembly. This takes the form of a Component Transfer Protocol (CTP). This is a high-level protocol for the assembly and disassembly of systems from sub-systems. It is a natural consequence of the architectural principles of the VSA: autonomy and adaptation; recursion and hierarchy; and invariants and self-reference. Specification of the CTP rests on the unique component structure and the nature of the interfaces as described above. Note that we have not implemented this protocol as of this writing, so it is notional at this point.

There are three phases or steps in the CTP: finding a component, joining and departing. Finding a component is the process whereby a component queries for a desired component based on some performance requirement. This assumes the existence of infrastructure support such as Microsoft’s Active Directory, OMG’s Trader, Jini or similar. How can a software component “know” how to request the services of another component? One way is to hard code in a query description for a particular type of component and provide some flexibility via runtime parameterization. An example would be specification for a color postscript printer with parameters of network domain and building number. Depending on the

application, the VSA provides support for more advanced query formulation. Inherent in the nature of control is that the controller must have a model of what it is controlling (i.e. the plant). The principles of adaptation have been described above. Within the component specification, the Planning function is most closely associated with modeling. Thus the architecture provides (requires) modeling of the system itself. This can be leveraged in support of finding components. Now we focus on the joining part of the protocol assuming a candidate component has been located.

The joining phase consists of the steps necessary to establish each of the interfaces of the viable component. (Numbers refer to the interfaces in Figure 4.) The first step is transfer of policy and rules from the acquiring controller’s Executive into the lower level component’s Executive. If the lower level component can comply with these, it signals by registering its direct alerts. (If the component cannot comply with the policies of the upper level controller it may be rejected.) This establishes interfaces 5 and 6. Next, the lower level component transfers its self-model (from Planning) to the upper level component’s Planning function. The upper level component can then process this information in order to incorporate the model of the new component into its self-model. This process may require the interaction of its 5-4-3 functions and result in changes to current plans, etc., that possibly affect the entire assemblage, including feedback to the lower level component across interfaces 5,6 and 4. Now interface 3 can be put in place. This is the Operations-to-Operations agreement. The upper level controller may grant resources needed by the lower level component and set reporting and accountability requirements (e.g. status reports needed for control). Next interfaces 2 and 7 are established. The production schedule (if needed) is transferred via 7 into the receiving component’s Regulator. Interface 2, Audit, performs its tests to verify the component is ready to start operating. Any connections between Plants are established (8 and 2’). Finally, the environmental interfaces (0 and 1) are established. For example, 1, which is the Planning to Environment interface may be a data set describing some aspect of the environment or a source for real-time information on the environment needed for planning at that component’s level.

The departing phase is just the reverse of the above and results in a component being released from its embedding framework. The process ensures that all other components are appraised of this just as they are during the joining phase.

4. From Architecture to Implementation

We are currently implementing the scenario described below. Our goal is to apply the high-level architecture of the VSA to a specific problem. The domain is that of business-to-business (B2B) e-commerce. We begin with a scenario and then go into detail to illustrate how we expect the VSA to support a component framework for B2B e-commerce.

4.1 B2B Scenario

Our scenario is based around three parties¹:

- **Muzac.com:** A producer of music in the form of MP3 files that are sold over the Internet.
- **eShop.com:** A provider of portal and retail (B2C) e-commerce site services.
- **B2B.com:** A B2B service provider that offers standard business documents, message formats and management software for B2B. *This software is based on the VSA.*

Muzac.com wishes to use eShop.com's portal and B2C services to sell its MP3 format music files to customers. Both Muzac.com and eShop.com are users of B2B.com's software services. This permits them to easily enter into a trading partner agreement based on B2B.com's standard business documents such as contracts, purchase orders, etc. They also use B2B.com's services for monitoring and enforcing the contract. Therefore, eShop.com and Muzac.com have already download standard business documents from B2B.com, customized them and integrated them into their operation.

We begin the scenario with a manager (human) at Muza.com filling out and signing a contract for portal services on the eShop.com website. The B2B.com software running on eShop.com's server routes it to B2B.com to start the contract implementation process. The contract is stored in B2B.com's document repository. B2B.com's contract management software then assembles a set of Business Policy documents that govern the behavior of all parties to the contract. These policy documents are generated automatically from the contract instance and a rule base of standard contract rules. The assembled policy documents are then transmitted to Muzac.com and eShop.com. The policy documents contain the set of rules each party must follow. These rules specify the obligations, prohibitions and permissions for each party relative to the contract.

The policy documents serve as the basis for generating a Plan Document. The plan is a set of activities specific to that party's performance of the contract. Finally, the

¹ Use of the names Muzac.com, eShop.com and B2B.com imply no relation to any real companies.

activities are mapped onto business object (attributes, methods and events) to implement the actual behavior in software. Also, as a result of this setup process all the necessary logic is in place for monitoring and enforcement of the contract. It is at this point that business transactions may begin. For example, Muzac.com may now send a purchase order for services, and begin transferring MP3 files into its allocated storage area on eShop.com. eShop.com will generate invoices based on usage. All of these transactions will be monitored by software running at each party's site as well as by B2B.com.

4.2 B2B Framework

The above scenario provides a domain setting for us to describe the implementation of B2B.com's e-commerce management software using the VSA. As a first step we develop a domain-specific software architecture for B2B. This is a focused component framework specification base on the terminology, processes and business objects of the particular domain. This process should translate the abstract high-level VSA into something engineers can readily apply.

We begin by mapping the names of the component functions into more familiar B2B and software related terms. We are fortunate here in that the VSM, on which the VSA is based, is an "enterprise" or human organization model. The only names we need to change are: Controller becomes Manager; Regulator becomes Scheduler; and Plant becomes Server. The term Organization will be used to refer to a component at the trading partner level. This is the viable system composed of a Manager and a Server. (The Controller and Plant in VSA terms.) Thus the B2B.com software running on the eShop.com and Muzac.com machines comprise an Organization from software identification purposes. We retain the names of the component interfaces from Figure 4. We will, however, refer to them as Trading Partner Protocols (TPP). The overall set of nine TTP (component interfaces) will be called the Trading Partner Agreement (TPA). The Component Transfer Protocol that specifies how component systems are assembled will be named the Enterprise Federation Protocol (EFP).

The Trading Partner Protocols (Figure 5 Interface Structure) are based on the Stability Criteria derived from the contract described in the scenario. The TPP will be expressed as policy documents (business rules) that eventually map to method calls on business objects. A System Element is a software component or an object instance (inside a component) that performs one of the Manager or Server functions. Transducers permit transformations between documents of the same format but having different schemas and between documents of different format types. For example a "Mapper" maps

between two XML documents with different schemas. A Translator translates between different formats such as XML, UN/EDIFACT and X12. Encryption and decryption codes are other examples of transducers. The Channel is of course the transport mechanism. In the case of B2B these might include HTTP, HTTPS, SOAP, etc. in the Internet, and MSMQ, DCOM, etc. in the LAN case.

We could continue in this manner to develop B2B domain specific terminology and tools. Some might include: a UML restatement of the VSM diagrams or an extension; a design methodology tailored to the VSA; and a reference implementation on an e-commerce environment such as Microsoft's BizTalk Server. However, we have a sufficient domain mapping to permit illustration of the major aspects of the VSA within the B2B scenario.

4.3 The B2B Framework in Action.

Now we return to the scenario and use the terminology and framework specified above to show how the features of VSA support the complexity of the B2B e-commerce domain. This section is organized as follows. The first step is to establish the "virtual enterprise federation." The behaviors of the three parties, the federation, are governed by the contract for services. The Enterprise Federation Protocol (EFP) sets up the federation based on the terms and conditions of the contract. This results in implementation of the overall Trading Partner Agreement (TPA). That is, the nine Trading Partner Protocols (TTP) between eShop.com and Muzac.com (sub-systems), and B2B.com (overall Manager). The next step is to describe the federation in operation. The goal is to show how the framework supports the range of operation from normal or routine, to adaptive through to supervisory-adaptive behavior.

4.3.1 Federation. We have done considerable work in implementing B2B contracts and have found that real world contracts are extremely complex documents [11]. From a control perspective they present a very large set of variables. Here we must use a reduced set of control variables. Accordingly we describe a simplified contract for services. Recall, Muzac.com produces MP3 music files and eShop.com supplies disk space on its server. Muzac.com is the *Purchaser* and eShop.com is the *Supplier*. Below are the clauses of the contract:

1. The contract has a start date and an end date.
2. Purchaser shall use the Supplier's price list to make purchases. A purchase specifies (a) quantity of service and (b) quality of service.
3. All payments shall be in AU\$.
4. Purchaser must issue a purchase order within 7 days of start of contract.
5. The billing period is every 30 days.

6. Purchaser shall make payment within 7 days of receiving an invoice.
7. Penalty for non-performance: Purchaser may fine Supplier \$1000 per hour of non-service; Supplier may fine Purchaser 10% of outstanding balance per day on any overdue payment.
8. Termination: Purchaser or Supplier may terminate contract upon violation of any clause.

As was stated earlier, both Muzac.com and eShop.com have installed and are running B2B.com's management software. The action begins when a contract arrives at B2B.com from eShop.com. Each clause in the contract has a set of rules (policy) associated with it. (These are stored in B2B.com's document repository.) The contract contains the particulars (e.g. names, addresses, period of service, etc.) and these variables are combined with the rules to make them specific to this contract instance. Based on these specific rules, B2B.com starts the Enterprise Federation Protocol (EFP) to setup the Trading Partner Agreement (TPA). We will simply refer to these rules sets as R1-R8 (corresponding to the contract clauses).

We now step through the EFP as it sets up each TPP for both the Purchaser and Supplier. Remember this is the protocol that assembles Muzac.com (Purchaser) and eShop.com (Supplier) into trading partners, with B2B.com as the overall contract manager. Referring to section 3.2.2 and Figure 4 may be helpful.

Setting up the TPA for eShop.com:

TPP5: The contract policy (rules sets) that specifies what behaviors eShop.com is obligated, forbidden or permitted to do are transmitted from B2B.com. eShop.com "internalizes" the policy. It goes through the Executive-Planning-Operations cycle to implement R1-R8. That is, each of the rules results in a planned activity being placed in the Scheduler, setting attributes on business object, and so on. For example, R1 and R4 set the start date of the contract as a trigger in SQL Server that will fire on the start date to check if a purchase order from Muzac.com has been placed in the database within 7 days of that date. If all the rules can be implemented with out violation of local policies, or other rules in place due to other contracts, then eShop.com signals B2B.com that it has accepted the contract.

TPP6: This is the agreement as to which direct alerts or error condition notifications will be sent directly to B2B.com's Executive. In this case, any condition that violates the contract will result in B2B.com be notified immediately.

TPP4: eShop.com transmits its self-model to B2B.com. For example, an ActiveX DLL that is a simulation of eShop.com's current machine configuration and one year of actual usage data. B2B.com incorporates this as described in 3.2.2.

TPP3: B2B.com's Operations function instructs eShop.com's Operation function on reporting requirements: weekly summaries of usage statistics, weekly report of Muzac.com's usage and monthly summary of Muzac.com's account balance and payments. eShop.com translates these recurring reports into its production schedule as activities that are lodged in its Scheduler.

TPP2: There are no scheduled activities received directly from B2B.com at this time. Initial activities have been setup as described in the last step. This TTP will be used in operation however.

TTP7: B2B.com performs a set of tests on eShop.com's server to verify that it is working correctly and to obtain data that is used to corroborate the data from the self-model of TTP4.

TTP8 & TTP2': This part of the protocol establishes the linkage between eShop.com and Muzac.com. TTP8 results in an account being set up for Muzac.com on eShop.com's site. The login and password are transmitted, storage space allocated, etc. TTP2' sets up the coordination channel between the two servers; the URL for posting activities is transmitted to Muzac.com.

In terms of setting up the TPA for Muzac.com the above description should give a good idea of how this process works. Both of these processes are going on in parallel but some synchronization is required. There is direct coordination between eShop.com and Muzac.com during the TTP8 & TTP2' set up. When Muzac.com receives the URL for posting coordination information it already has its weekly production schedule of MP3 files lodged in its Scheduler. It transmits this output schedule information to eShop.com as a courtesy. This can be used by eShop.com in its own Manager (controller self-model) to anticipate storage load requirements and communications demands. We see here the possibility of a complex coordination feedback. For example, eShop.com might offer Muzac.com a discount to change its deposit schedule if it happens to fall within a particularly high demand period.

We have not mentioned mappers and translators. These correspond to "transducers" in the Figure 5. For example, the system we are currently using, BizTalk Server [10], provides tools to define mapping between a wide range of message types (e.g. EDI, EDIFACT, X12). These can be placed in the interface statically or dynamically based on message content.

With this, the EFP is complete. The TPA for eShop.com and Muzac.com are set up and the parties are ready to begin business operations. Figure 3 is a suitable diagram of this configuration.

4.3.2 Business Operations. We can now exercise the federation and examine its behavior over a range of conditions starting with normal (equilibrium) operations.

Based on the particulars (variables) of the contract that was filled in by the (human) manager of Muzac.com, a purchase order (PO) can now be transmitted to eShop.com. This should be within 7 days of the contract start date and specify a quantity of service and a quality of service (R1-R4). Assume a PO is sent based on eShop.com's price list for 20 Gigabytes of storage and the highest quality of service (zero down time and daily backups). The PO is transmitted to eShop.com and an acknowledgement is received via TTP8. Now Muzac.com may at any time begin depositing its MP3 files on the portal. This is also via TTP8. eShop.com sends an invoice for payment at the end of the billing period (R5). Muzac.com makes an electronic payment within 7 days (R6). Again, these are in accordance with TTP8. Both eShop.com and Muzac.com are sending regular reports to B2B.com (TTP3) so that it can perform its monitoring services. These status reports also carry information needed by B2B.com's Planning function; it uses this information as input to fine-tune its simulations. Planning is constantly projecting forward in time to see if eShop.com will run out of storage space, for example, or to advise Muzac.com that it should send a PO to request a larger storage space. If Muzac.com is consistently underutilizing its storage, B2B.com might advise that it reduce its storage space to save money. Also, B2B.com randomly checks on both parties via TPP7. For example, it sporadically accesses Muzac.com's web page on eShop.com's server and downloads a file to verify the server is running and that download speed is adequate. As was mentioned earlier, both parties might coordinate closely (TTP2') to further optimize their business operations.

The above gives a picture of normal operations with some minor adaptive optimization. Both parties are well within the behavior boundaries of the contract (R1-R6) and B2B.com is performing its monitoring role. Now we push the system away from equilibrium to see how it can retain stability. We will explore several possible problem areas that might normally arise in such an e-commerce setting.

A common problem would be that eShop.com's server goes down due to a software/hardware failure. B2B.com would detect this (TTP7) and inform both parties (TTP3). (Muzac.com might detect the server down first and notify B2B.com's Executive via TTP6.) B2B.com would instruct Muzac.com to delay its scheduled MP3 deposits until further notice. Clearly eShop.com is in violation of the contract (R1-R6) and Muzac.com has a remedy: it can fine the supplier and/or terminate the contract (R7 and R8). This might be a situation where humans should be involved. This is determined by the policies in B2B.com regarding notification. But for now assume the policy is for B2B.com to try and solve the problem itself before notifying anyone. First, B2B.com's Operations asks Planning for eShop.com's mean time to repair on failure

data. If the time is short, a few hours, nothing is done and the situation is monitored. If the server is not back within this mean time to repair, other strategies can be employed based on policy. Human managers at both parties would be sent email with the details of the outage. After some period of no service, B2B.com would begin emailing (legally binding) fines to eShop.com. It might also begin searching for candidate replacement sites to recommend to Muzac.com's management. If Muzac.com's management really trusted B2B.com's ability it might have authorized it to terminate a supplier in violation of a contract and automatically enter into a contract with a new one.

Based on this experience (bad service from eShop.com) both B2B.com and Muzac.com (planning function) refine their profile (model) of eShop.com. Planning can modify the rules by which Operations deals with eShop.com. This is the learning process of adaptation. Muzac.com can use this information when searching for other service providers.

Finally, consider that the B2B.com Manager for this account is a sub-system of a larger e-commerce federation. There is a component in that federation - BestPortalBuys.com - that regularly surveys, categorizes, and recommends portal service sites. B2B.com subscribes to (has a TPA with) this service, passing the cost on to Muzac.com of course. These ratings would come in handy in the example above. They could also be used by B2B.com to dynamically acquire new portal services and discard existing portal providers for Muzac.com if policy permits.

As for the supervisory level of functionality, I think humans will continue to fulfill this role. They will set policy in the form of rules that are simply implemented. It is not yet clear how to develop systems that generate their own policies or if it is even desirable.

5. Conclusion

In summary, we have presented one approach to tackling the hard problems posed by complex systems and their software application requirements. We chose the Viable System Model as a starting point only; our goal is not to verify or validate the model, or to follow it slavishly. We see it is simply as an instance of a model from the disciplines of general system thinking and cybernetics; disciplines that seek to understand complex systems. Our hope is that the notion of software viability and the architecture we describe may contribute to the development of better software. We are currently developing a reference implementation of the architecture based on the B2B scenario described above.

References

- [1] *IEEE Computer Society, Technical Committee on Complexity in Engineering*, <http://www.elet.polimi.it/tccx/>.
- [2] *6th IEEE International Conference on Engineering of Complex Computer Systems (ICCECS 2000)*. Boissonade Tower, Ichigaya Campus, Hosei University, Tokyo (Japan): IEEE Computer Society.
- [3] Beer, S., *Diagnosing the System for Organizations*. 1985, Great Britain: John Wiley and Sons Ltd.
- [4] Shaw, M., *Beyond objects: A software design paradigm based on process control*. ACM Software Engineering Notes, 1995. **20**(1).
- [5] Herring, C. and S. Kaplan. *Cybernetic Components: A Theoretical Basis for Component Software Systems*. in *Component Oriented Software Engineering Workshop (COSE'98)*. 1998. Adelaide, Australia: (<http://www.dstc.edu.au/TU/staff/herring/cose98-ref.pdf>).
- [6] Szyperski, C., *Component Software*. 1998, New York: Addison-Wesley.
- [7] Passino, K.M. and S. Yurkovich, *Fuzzy Control*. 1998, Reading, Massachusetts: Addison-Wesley.
- [8] *The Community Resource for Jini Technology*, www.jini.org.
- [9] *Universal Plug and Play Forum*, <http://www.upnp.com/>.
- [10] *Microsoft BizTalk Server 2000*, <http://www.microsoft.com/biztalkserver>.
- [11] Goodchild, A., C. Herring, and Z. Milosevic. *Business Contracts for B2B*. in *Infrastructures for Dynamic Business-to-Business Service Outsourcing (ISDO'00)*. 2000. Stockholm, Sweden.