The Viable System Model for Software

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

We are investigating the application of Beer's Viable System Model to the development of software for "complex" applications. We use this model as the basis for a component system architecture, the Viable System Architecture. A key construct of the architecture is the structure of a component and its special set of interfaces, a "viable component." After presenting the Viable System Model and the Viable System Architecture, we outline a design methodology for development of software components in this approach.

1. Introduction

Stafford Beer developed the *Viable System Model* (VSM) [1] for the purpose of understanding, predicting and controlling human organizations or "enterprises." The VSM is based on study and observation of many organizations over a thirty-year period. Beer's goal was to discover the invariant structures and behaviors and describe them based on cybernetics. We take this model as a starting point for modeling "complex" systems in order to build better software.

Examples of complex software systems include Smart Environments, Ambient Computing, Multi-Agent Systems, Adaptive/Intelligent User Interfaces and Business-to-Business e-Commerce. These 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.

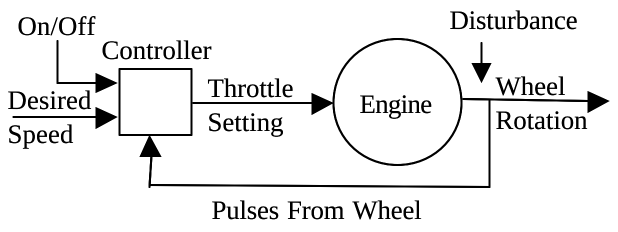

Figure 1 Automobile Cruise Control

As an introduction to and motivation for the approach consider the automobile cruise control system of Figure 1. This is a typical closed-loop control system where the controller monitors and maintains a system variable, speed, within certain desired limits. The overall system is subject to environmental disturbances (such as a steep hill) and the controller, based on feedback, adjusts the throttle as necessary.

The automobile cruise control is one of the "canonical" object-oriented design problems. Shaw analyzes this problem and develops a *control paradigm* for software [2]. Her insight is that *it is a control system* and there is a body of engineering knowledge associated with such systems. She argues that the object-oriented paradigm may not be the most suitable approach in this case. She develops a software architecture based on classical feedback control theory as opposed to an architecture based solely on identification of objects and their relationships.

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 the (domain neutral) object-oriented approach. In particular, performance and correctness constraints are identified early in the design process.

Shaw offers the following "rule of thumb" for use of the 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 [3].

Therefore, after considerable search and evaluation, we chose the VSM as the basis for development of the *Viable System Architecture* (VSA) [4]. 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. The VSA architecture defines a unique component structure, the "viable component" and a set of component interfaces that in tern defines the framework itself.

In this paper we concentrate on the structure, design and implementation of the fundamental component. We begin with a brief introduction to the VSM. Then the VSA is presented as a component framework. The desired qualities of software developed based on this approach are stated. The principles that lead to these qualities are described. The internal structure of a component and its interfaces is given in detail. Finally, a methodology or sequence of steps for designing viable components is outlined.

2. The Viable System Model

The following is a brief introduction to the features and structure of the VSM. 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.

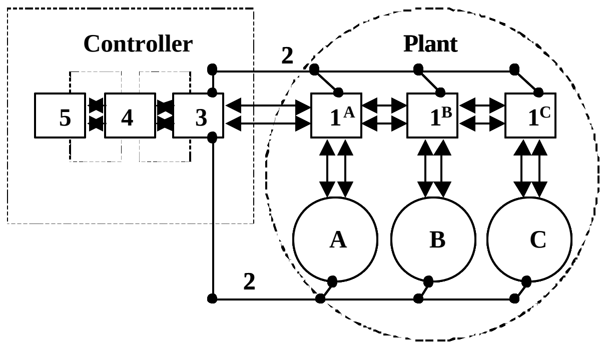

Figure 2 Simplified VSM Diagram

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^{A-C} 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 [5]. Autonomy is the degree of freedom the level 1^{A-C} 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. 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.

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. 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 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.

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.

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. 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.

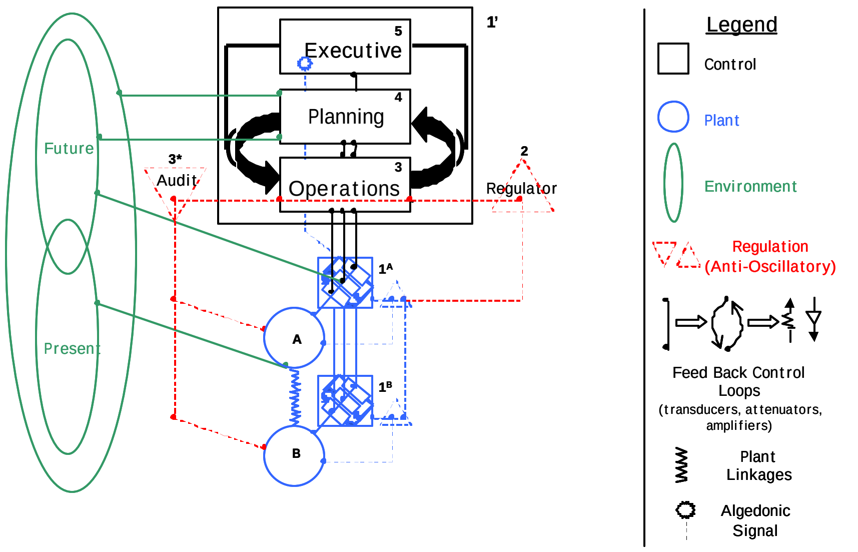

Figure 3 Viable System Model Diagram

4. The Viable System Architecture

The preceding sections have hopefully 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^o 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.

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

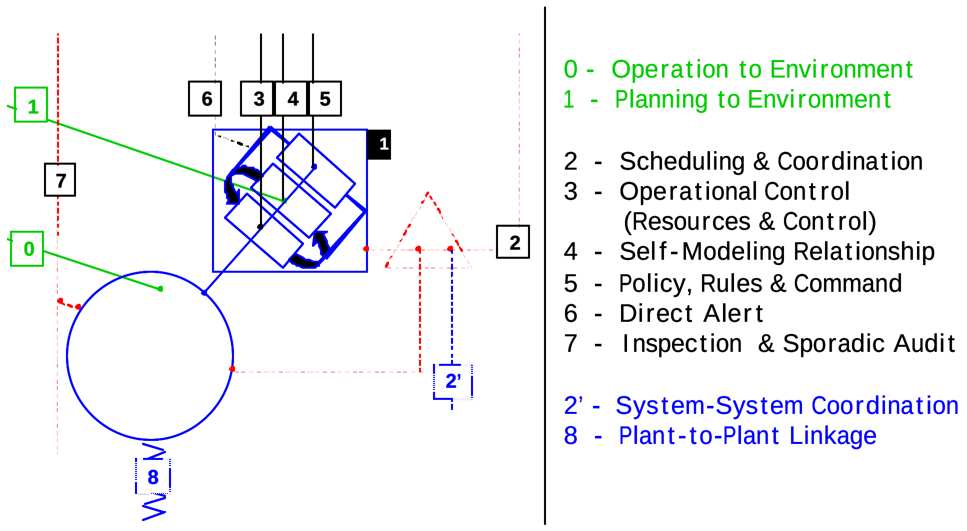
1
6
3
4
5
1
7
0
2
2'
8
0 - Operation to Environment
1 - Planning to Environment
2 - Scheduling & Coordination
3 - Operational Control
(Resources & Control)
4 - Self-Modeling Relationship
5 - Policy, Rules & Command
6 - Direct Alert
7 - Inspection & Sporadic Audit
2' - System-System Coordination
8 - Plant-to-Plant Linkage

Figure 4 Viable Component Interfaces

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:

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.

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.

Traditionally, software components are developed in isolation. They are distributed with an interface specification and it is the job of the user (programmer) to wrap, glue or otherwise integrate the component into the intended system. The VSA component framework defines a standard pattern of interfaces on a component. This permits design with knowledge of the components on both side of the interface. The goal is to achieve dynamic equilibrium – homeostasis – for each set of coupled components.

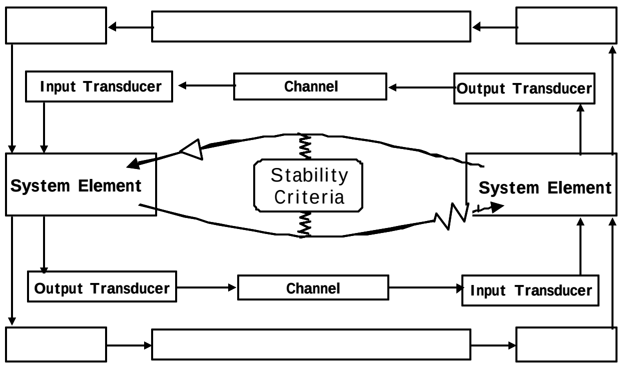

Figure 5 Interface Structure

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.

Finally, it should be noted that interfaces are not designed in isolation. For example, suppose three plants are to be connected to each other. In this casethe stability requirements must be developed with all interfaces in mind, or the design refactored, as new interface requirements are discovered.

5. Designing Viable Components

In this section we describe a methodology, a sequence of steps to design software systems based on the VSA. These steps are a synthesis of approaches given in Beer [1], traditional control [6] and aspects of fuzzy control [5]. The goal is to outline a method of designing a component at one level in a recursive hierarchy of sub-systems. First it is necessary to choose a level at which to start. This choice will be determined by application requirements. For example, our task may be to assemble a set of viable components into a new viable software system for some stated purpose. In that case we would have, on hand, software components built according to the VSA framework specification. That is, a software component with a set of interfaces as described above and shown in Figures 4 and 5. These can be treated as "black box" components. In this case the set of interface specifications and a description of the behavior of the components provide all the information needed to design at the next level of the hierarchy. On the other hand, we may be starting at "ground level." Here we are given a plant, e.g. a digital camera or a file server, and the task is to design a controller conforming to the VSA framework so that it may become a viable sub-system within our overall system architecture. We describe this design case next.

The first step is to identify the "System-In-Focus." That is, clearly define and give a name to the system for which the controller is being built. The meta-system or type of meta-system the "System-In-Focus" will be embedded in is usually known. (The system in focus is usually embedded in many meta-systems.) Also, clearly define systems that the system in focus embeds, if any. It is helpful to use a diagramming technique such as in Figure 3. Referring to that figure, we choose "1^A+A" as the system in focus for the remainder of this section. Our goal now is to build the controller, "1^{A}", for the plant "A." We name the system in focus *System A*.

Having identified the system in focus, *System A*, a detailed analysis is conducted. The purpose of this analysis is to determine the performance requirements of *System A* in relation to its meta-system and environment. The analysis describes the major subdivisions of the system: Plant, Controller and Environment. The overall performance requirements of *System A* are specified as are the information flows between its main subdivisions. This includes identification of sensors and actuators. The result of this step should be a model and a simulation of the system. The modeling techniques chosen depend on the nature of the plant. These range from “hard” to “soft” system approached. Hard systems can be described explicitly by equations whereas soft systems are described using heuristics (e.g. rule bases). Usually a mixture of the two is needed.

With an understanding of *System A*, its functional requirements and its simulation, a detailed design of its controller can be accomplished. This design is necessarily constrained by both the plant and the meta-system in which it must operate. That is, the next two steps, controller design and meta-system interface design are interrelated. We describe interface design next.

The interfaces from *System A* to its embedding meta-system are determined (if existing) or designed as needed. These are exactly the nine interfaces described above and shown in Figure 4. Note there are three groups of interfaces to consider: *System A* to meta-system (numbered 2 through 7); *System A* to environment (0 and 1); and *System A* to other systems at the same level (2’ and 6). The nature of these interfaces is described in detail in the last section. Note, each of these interfaces are between system elements **within** *System A’s* controller and plant and system elements of other system as indicated in Figure 5.

The controller consists of the elements 2-5 as shown in Figure 3. The problem is now to design them. The order in which they are designed depends on the following considerations. If there are existing components at the next higher level, they may have a strong influence on the representation of fundamental constructs such as policies, plans, etc. In this case, examining the interfaces as describe above first may indicate a top down approach. That is, it may be easier to start with the Executive (5) function. Also, if there are systems *at the same level* with direct input and out connections, the design of the Regulator (2) for scheduling and coordination will be partially determined.

In the absence of known constraints, begin design of the controller with the Operations-Regulator-Auditor triad, as it is most closely associated with the plant. This corresponds to a “standard controller” in most controller design approaches. In the VSA, the Regulator and Auditor are distinguished because of the hierarchical nature of the framework. They have explicit links to similar elements with other controllers at higher levels and at the same level. It is therefore convenient to represent them as separate objects. Now, it must be stated that only so much general advice can be given on the design of a controller. The exact design is inherently dependant on the specific plant. General references on control are given at the beginning of this section. It has been our experience in domains such as smart environments and business-to-business, that elements of standard control theory, fuzzy control theory and rule-based (AI) approaches are usually combined to achieve a solution. Bearing this in mind, general guidance is provided.

The Operations function exercises immediate control of the Plant. It does this via the sensors and actuators identified in the modeling phase. In design phase, the simulation is all-important. The simulation specifies what control variables are available for manipulation and is used to prototype the Operations module. A range of basic commands is usually implemented such as Start, Stop, Pause, Reset, etc. In terms of monitoring the plant, if possible, a software layer in or on top of the plant provides routine status reports in an appropriate format for Operations and the other control functions. These status reports (on the internal variables of the Plant) are the chief source of information on which Operations’ control algorithm will be designed. We have found a rule-based approach offers considerable flexibility in implementing these algorithms.

Operations directs both the Regulator and Auditor. The main function of the Regulator is to implement a *Plan*. A plan is a schedule of activities for the Plant. The Regulator also coordinates the plan with components at the next higher level and at the same level. It is necessary to determine a representation of the plan and its activities relative to the possible actions of the Plant. A template for representing activities we have found useful is:

Activity #: <Object> <Action> [Before <Condition>] | [After <Condition>] | [Satisfy <Condition>] [, If <Condition>] [, Where <Condition>][, Otherwise Activity <#>].

Regulation is dynamic as plans may be frequently modified. Again, implementing the Regulator using a rule-based approach may help or a custom scheduler can be written. In some systems a Regulator may already be present in some form and it can be used directly. For example, most databases provide for workflow modeling. It may also be the case that the Plant itself contains a scheduler and using it may be the simplest approach.

The Auditor performs inspections and checks to provide verification and assurance to Operations that the Plant is functioning properly. This is in addition to and independent of the normal or routine status reporting by the Plant. The controller at a higher level in the hierarchy may also place conditions for auditing certain plant variables. A key benefit of the Auditor is that it can be set to inspect control variables at sporadic or random times. The information derived is used to inform Operations in between the Plants’ routine reporting cycle. In implementing the Audit function we have found it useful to also provide a separate Accountant function for the Plant if it does not already have one. The Auditor can be directly connected to this Accountant.

Planning performs two major functions: "tuning" Operations and transmitting commands from the Executive to Operations. While Operations is concerned with internal control of the Plant; Planning anticipates future conditions in order to inform Operations. Planning does this by modifying Operations' control algorithms, for example, changing rules in its rule base. This will be reflected in changes to the activities scheduled by the Regulator.

Planning must have a model in order to accomplish its function. This may take the form of data sets based on previous runs of the Plant. This approach permits application of statistical methods to detect trends that can be used to improve Operations' control algorithms. Planning may also make use of the Plant simulation developed earlier. The simulation can take advantage of real-time data from the running Plant as well as data obtained from the Environment to aid Operations. The second major function of Planning is to mediate between the Executive and Operations functions. Commands issued by the Executive are filtered, translated and possibly re-ordered based on Planning's current model of the Plant.

Finally we can design the Executive function for *System A*'s controller. The Executive serves to provide overall supervision of the Planning-Operations functions. That is, it bounds the possible range of actions of the lower levels of control. We choose to express these governing conditions as policies. For example, a language for specifying policy as a set of rules is [7]:

Rule #: <Role> [is] (Obligated | Forbidden | Permitted) [to] [do] (<Action> [Before <condition>] | Satisfy <condition>) [, If <condition>][, Where <condition>] [, Otherwise see Rule <#>].

Policy is a synthesis of policy transmitted from the higher-level controller and local policy. The Executive must translate the synthesized rules into instructions and plans for implementation by Planning and Operations. Note, the terms *obligated*, *forbidden*, and *permitted* make the above policy language a declarative specification. This form of rule can be used to prove assertions about system behavior. Also, declarative rules may be mapped into the (imperative) plans and activities as described above. Again, rule-based programming approaches are appropriate for implementing these types of knowledge representations.

This completes the description of developing a viable component starting from an existing Plant. Now we return to the other design problem mentioned at the beginning of this section, that of assembling existing viable components into a system. There are two possibilities: designing a new component (at the next higher level) to control existing components and integrating exiting components into an existing system. In the first case the approach is similar to the steps outlined above. That is, a new controller is designed based on existing components that provide viable component interface specifications. In the later case, existing viable components are to be assembled into an existing framework. This may require extending components at both levels: the difficulty of this task depends on how flexibly the components are implemented. It may be possible to introduce suitable transducers (as shown in Figure 5) to permit assembly of the components without changes. Finally, it is feasible to develop a protocol for dynamic assembly of viable components into systems. Development of such a protocol is facilitated by the unique interface structure of the components. However, protocols of this nature are domain specific and require consider experience in developing a range of components and frameworks based on these components.

6. Conclusion

In this paper we have shown how Stafford Beer's Viable System Model can serve as a reference model for the development of a software component system architecture to address the needs of complex systems. We call this software architecture the Viable System Architecture. The key contribution of the architecture is the unique structure of a component and its special set of interfaces, the "viable component." This unique component structure recursively defines the framework itself. We have outlined a design methodology for development of this class of components. The methodology is based on our experience in developing smart environments [8] and business-to-business systems [9]. We are currently developing a reference implementation of the architecture on a commercial business-to-business e-commerce platform.

References

1. Beer, S., *Diagnosing the system for organizations*. 1985, Great Britain: John Wiley and Sons Ltd.
2. Shaw, M., *Beyond objects: A software design paradigm based on process control*. ACM Software Engineering Notes, 1995. **20**(1).

3. 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).
4. Herring, C. and S. Kaplan. *Viable Systems: The Control Paradigm for Software Architecture Revisited*. in *Australian Software Engineering Conference*. 2000. Canberra.
5. Passino, K.M. and S. Yurkovish, *Fuzzy Control*. 1998, Reading, Massachusetts: Addison-Wesley.
6. Franklin, G.E., J.D. Powell, and A. Emami-Naeini, *Feedback control of dynamic systems*. 2 ed. Control Engineering, ed. T. Robbins. 1991, Reading, Massachusetts: Addison-Wesley.
7. Steen, M. and J. Derrick. *Formalizing ODP Enterprise Policies*. in *Proceedings of Enterprise Distributed Object Computing Conference (EDOC'99)*. 1999.
8. Herring, C. and S. Kaplan, *Component-based software systems for smart environments*, in *IEEE Personal Communications*. 2000.
9. Herring, C. and Z. Milosevic. *Implementing B2B Contracts Using BizTalk*. in *Thirty-Fourth Hawaii International Conference on System Sciences (HICSS-34)*. 2000. Maui, Hawaii: submitted.