Adaptable and Adaptive Systems: The Intelligent Control Paradigm for Software Architecture

Charles Herring
Department of Computer Science and Electrical Engineering
University of Queensland, Brisbane, Australia, 4072
herring@dstc.edu.au

Abstract

This paper presents a model-based architectural approach to evolving, or growing, adaptive software systems. The architecture is based on the Viable System Model (VMS) developed by Stafford Beer. The VSM, as a meta-model, tries to capture the essential invariants required by successful, or viable, systems. These types of systems are called Complex Adaptive Systems in Complexity Theory and Intelligent Control Systems in control engineering. I call the VSM-based architecture the Viable Software Architecture. The architecture incorporates the structure and behaviour of viable systems in software. Additionally, the approach permits "piecemeal" growth of the system to include high-levels of behaviour. The goal is to provide a method whereby software can be adapted (design-time, by humans) toward becoming an adaptive system (at runtime, self-controlling software). The Viable Software Architecture is developed in a sequence of stages using UML. Then a component framework specification is given based on the architecture. The overall goal of the approach is discussed.

1. Introduction

This paper presents a model-based architectural approach to evolving, or growing, adaptive software systems. The architecture is based on the Viable System Model (VMS) developed by Stafford Beer [1, 2, 3, 4, 5, 6, 7, 8]. The VSM, as a meta-model, tries to capture the essential invariants required by successful, or viable, systems. These types of systems are called Complex Adaptive Systems in Complexity Theory and Intelligent Control Systems in control engineering. I call the VSM-based architecture the Viable Software Architecture. The architecture incorporates the structure and behaviour of viable systems in software. Additionally, the approach permits "piecemeal" growth of the system to include high-levels of behaviour. The goal is to provide a method whereby software can be adapted (design-time, by humans) toward becoming an adaptive system (at runtime, self-controlling software).

Strictly speaking, the approach described here is an extension of the "Control Paradigm" developed by Shaw [9] and can be called the Intelligent Control Paradigm of software architecture. Shaw developed the "Control Paradigm" as one approach to the canonical design problem of the cruise control challenge [10]. She says:

"This system organisation [feedback control software] is not widely recognised in the software community; nevertheless it seems to *quietly* appear within designs dominated by other models. Unlike object-oriented or functional designs, which are characterised by the kinds of components that appear, control-loop designs are characterised both by the kinds of components involved and the special relations that must hold among them." [11 p.28]

Shaw describes the normal view of software as *algorithmic*: input-process-output. Further, software is rarely developed to handle any change in external environments or even input. In this sense, it can be characterised as an open-loop control system. This is appropriate for simple, static systems; but cannot be expected to cope with a dynamic or unpredictable environment. The position here is that most advanced software systems are affected by external disturbances and hence the control paradigm must be considered for at least some parts of the overall architecture. Further, adaptive control is feasible and may be necessary.

2. The Viable Software Architecture

The following sequence of UML diagrams develops the Viable Software Architecture (VSA). There are seven diagrams that represent the conceptual development of the model and the "piecemeal" evolution of a software system built according to this style. These steps are hereafter referred to as "stages." The seven diagrams capture the seven fundamental invariants in the VSM and their relationships. UML collaboration diagrams are used. Collaboration diagrams consist of ClassifierRole and their relations. ClassifierRoles are shown in a box with the following label format: /ClassifierRoleName: ClassifierName." A role is not an object; it is a classifier that describes many possible objects that may perform that role within an instance of the collaboration. The ClassifierName is

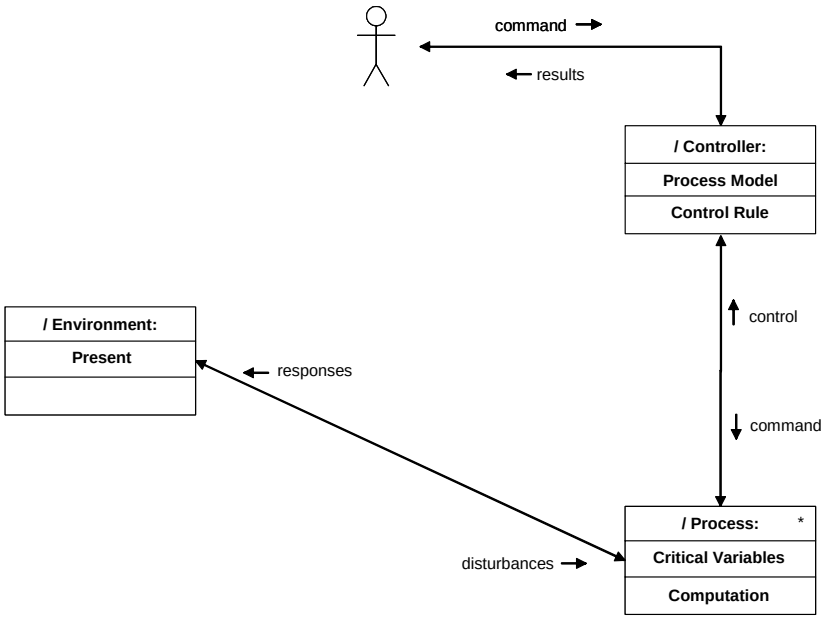

Figure 1 The Control Paradigm (Stage 1)

an optional element is not used in the abstract architecture specification.

Stage 1, the basic Control Paradigm architecture style is shown in **Figure 1**. The actor symbol represents a human or software agent that gives direct commands, "set points," to the system and observes the results, e.g. cruise control. This is the first step in transferring human control skills into "self-controlling" software systems. The controller is a simple feedback controller that implements the commands and monitors execution of the command based on "control" messages. This is the feedback-control information from the process. The environment is shown as the source of disturbance in which the overall actor-control-process system is operating. Note that more than one process may be under control (indicated by the * in the upper right corner of the ClassifierRole box).

Stage 2 requires the development of a separate regulator or regulatory centre. This is shown in **Figure 2**. This necessitates the explicit representation of plans, activities, time tables, etc. based on the application context. The development of this separate facility provides for regulatory coordination (anti-oscillation) among systems at all levels: the "regulatory system." Really, it is a sub-system within the viable system, not a separate system in its own right. Note there is a feed-back path from the process through the regulator back to the controller. Also note there may be more than one regulator for a process.

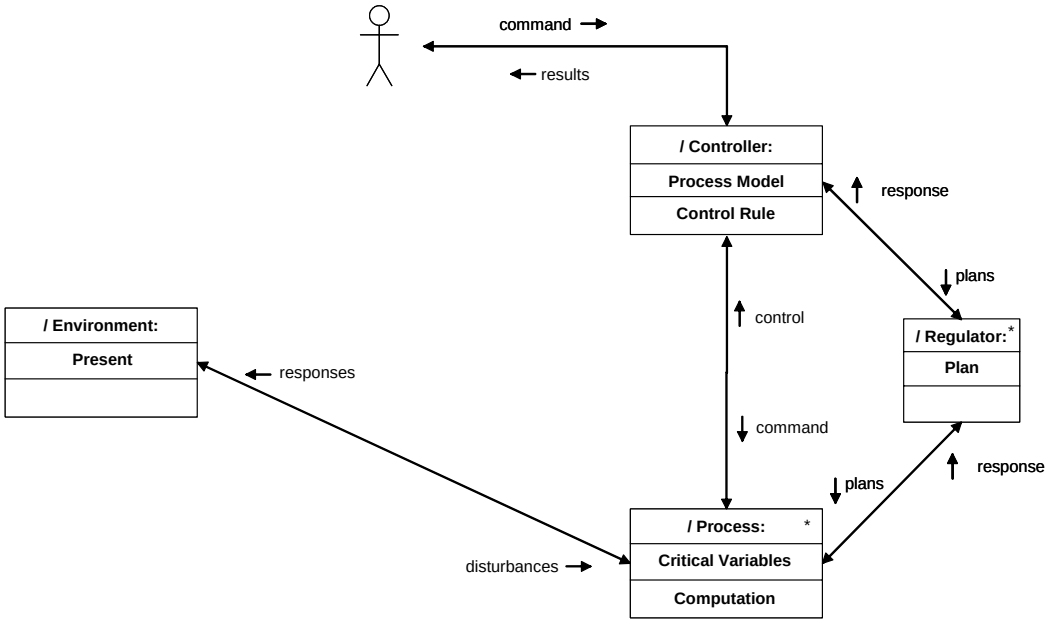

Figure 2 Separate Regulator (Coordination, Stage 2)

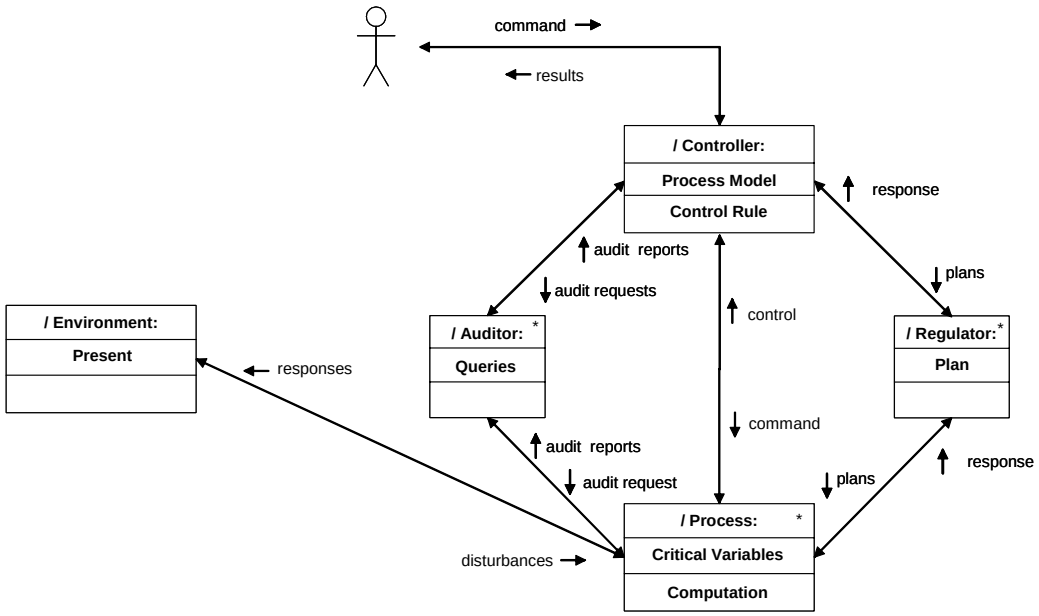

Figure 4 Auditor (Stage 3)

Stage 3 (possibly accomplished in parallel with Stage 2) sees the introduction of the Auditor as shown in **Figure 4**. In the case of multi-layered viable systems, this results in the "auditing system" across all layers. Note that more than one auditor may be necessary. Addition of this function will most likely require change in the controller. The direction of the auditor will come via the actor, and the information supplied will be of most interest to that agent.

The functionality of control is extended with an Adaptive Controller in Stage 4 as shown in **Figure 3**. This represents the second major step in migrating human control skills into a system. Note the Actor's input changes to a more abstract language. The term "rule" is used in the diagram above in the sense of condition-action pairs, or if-then statements as found in fuzzy control, expert systems and so on. Also, note the relationship the adaptive component has to the environment.

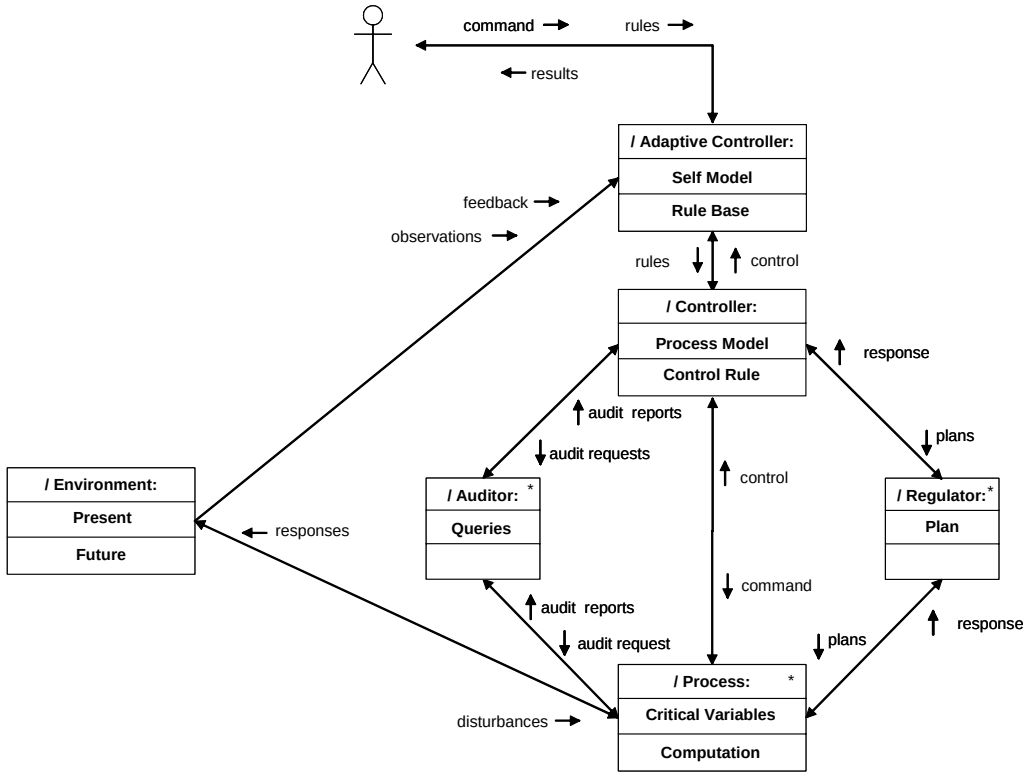

Figure 3 Adaptation (Stage 4)

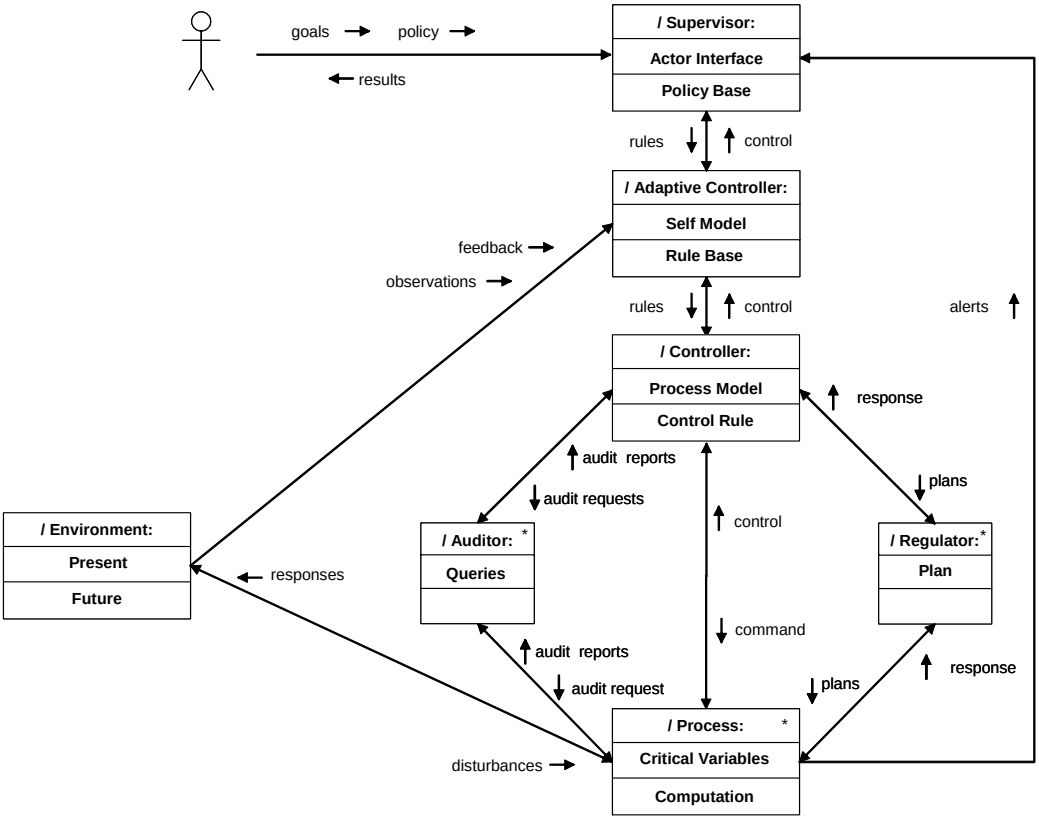

Figure 5 Supervision (Stage 5)

The abstract architecture is completed at Stage 5 with the addition of a supervisory loop governing the adaptive-control pair as indicated in **Figure 5**. Depending on the sophistication of the supervisor and its interface, the actor can now specify overall system behaviour in a declarative manner. The term “policy” is used here to mean a high-level, declarative statement on how goals are obtained. These statements govern or constrain the application of the rules, or control laws, of adaptive control. Also, note the addition of an association between the process and the supervisor over which alert messages are transmitted that bypass the lower levels of control.

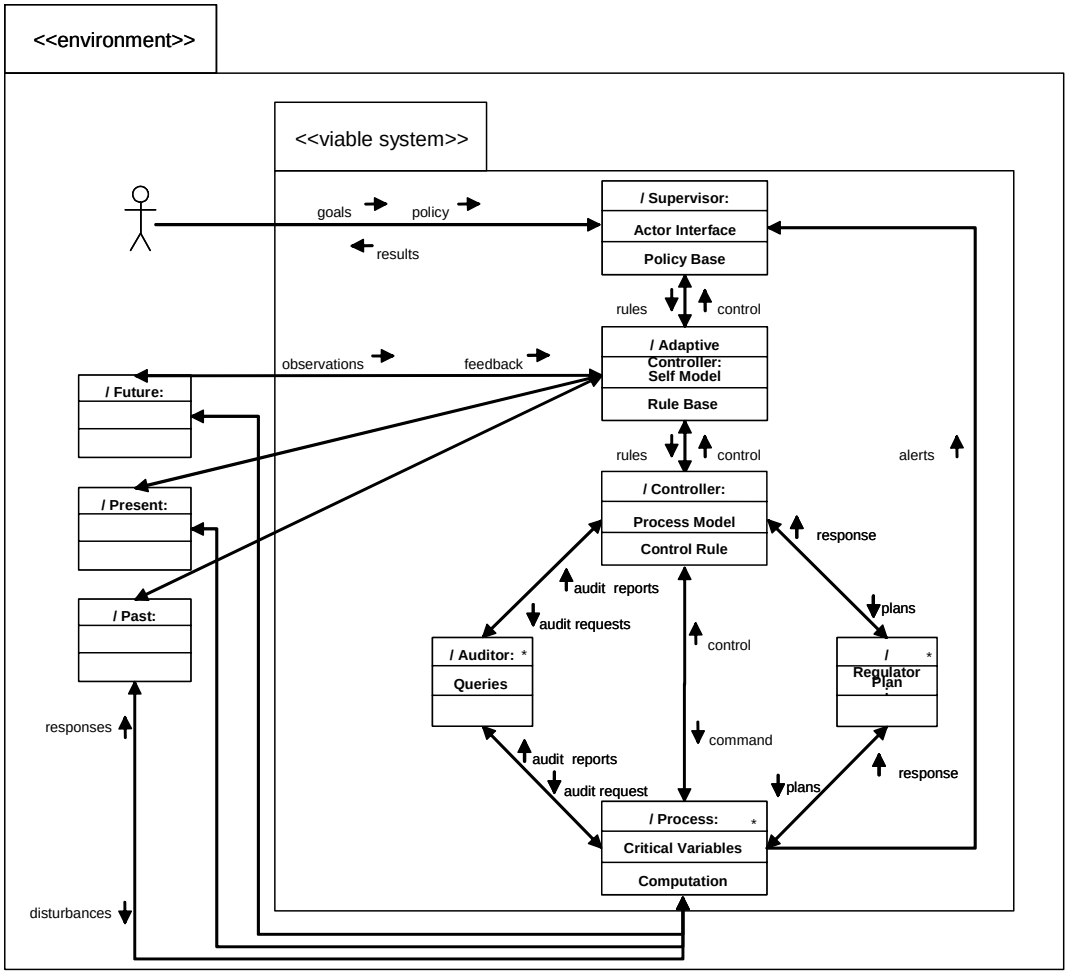

Figure 6 Composition of Systems (recursive, systems-of-systems)

3. From Architecture to Design: A Framework

This section develops an abstract component framework based on the VSA. This framework provides the interface structure of the characteristic "viable component" that will become the basis for design and implementation of domain specific frameworks. An important aspect of this component specification is that it is both the fundamental building block of the framework and it defines how systems are composed recursively.

The composition of systems based on the VSA is shown in **Figure 6**. The UML package diagram is used to show how systems can be composed of other systems in a recursive style. In the diagram, there are two levels of systems shown: larger package and the viable system embedded within it. This is a major feature of the architecture and made explicit in the framework. Systems developed based on this approach have a well defined relation, or set of interfaces that support the "system-of-systems" style of building large systems. Note the associations marked *Adaptive-Adaptive* and *Supervisor-Supervisor*, these indicate relationships between adaptive and supervisory controllers at the next level of recursion.

Figure 7 gives a detailed diagram of the associations between a viable software system and its two embedded systems. The associations are numbered and the nature of these associations will be discussed in detail. Conceptually, they fall into four categories:

0: actor to system (may not be present at every level).

1-6: meta-system to system (system to embedded system),

7: system to system, and

8-9: system to environment.

The results of the previous designs (**Figure 6**and **Figure 7**) culminate in the component shown in **Figure 8**. This Viable Component is the building block on which the framework is based. Every viable component must offer these interfaces as they permit systems to be related to other systems either by embedding or by coordination at the same level. The nature of these interfaces is based on the organisation of the VSM. This unique component interface permits the specification of a *component transfer protocol* for the dynamic assembly of systems.

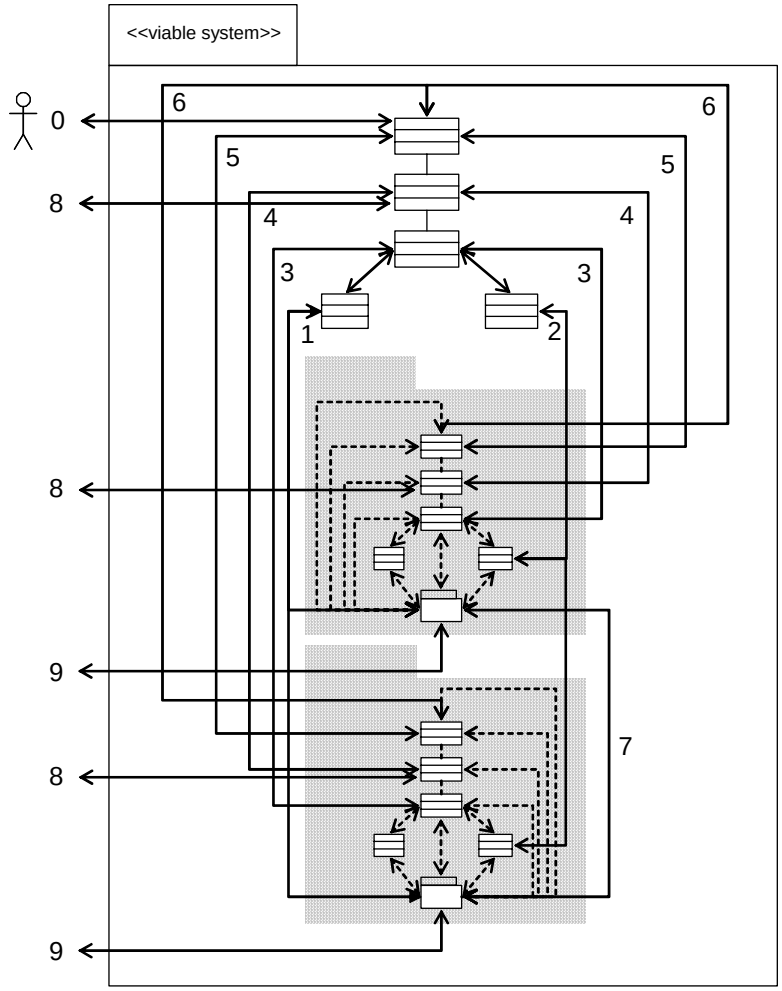

Figure 7 A System with Two Embedded Subsystems

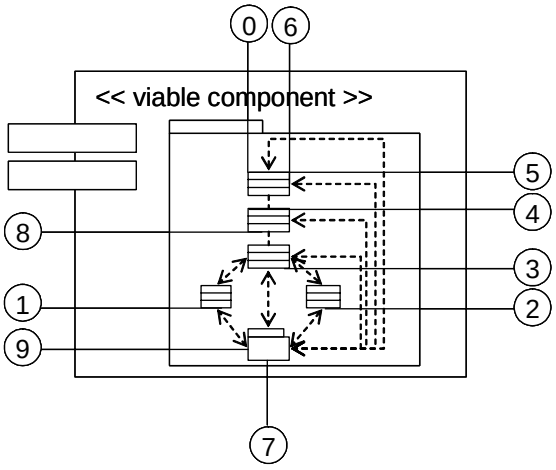

Figure 8 The Viable Component

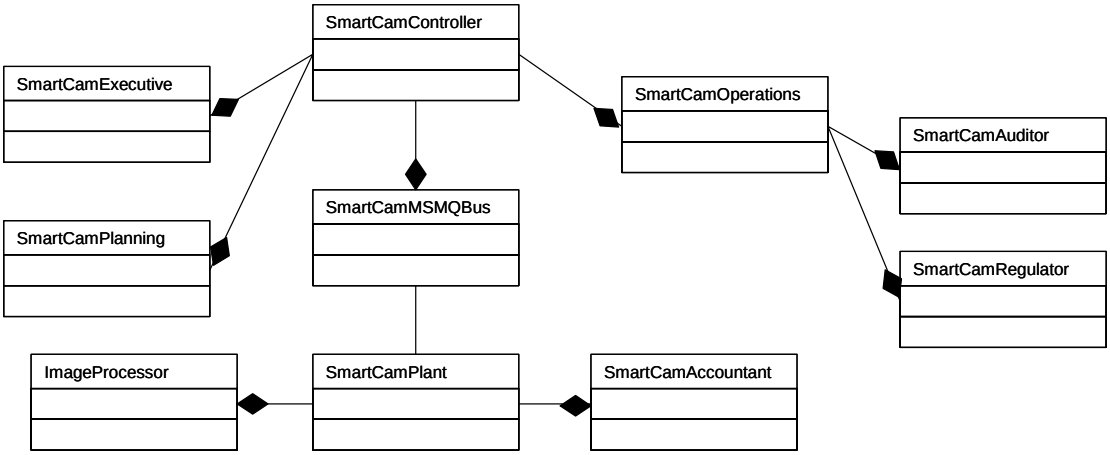

Figure 9 SmartCam Class Diagram

4. Case Studies

The VSA was applied to three domains-specific case studies and the results documented in [12]. The first study shows how the VSA can be used for analysis of existing systems. The *Groove* Internet-based, peer-to-peer, collaboration system was chosen, because its architecture was well documented, and because it represents a large class of interactive systems. Groove's architecture is mapped into a Viable Software Architecture and extended with an adaptive user interface feature by introducing a Fuzzy Controller.

A second case study focuses on e-Commerce with a B2B scenario outlining how the VSA supports adaptable system growth. The area of "electronic contracts" for B2B was explored in depth and a prototype implementation developed using an Expert System (Visual Rules Studio) to build an adaptive control.

The third application explores the emerging area of "Smart Environments." A scenario was developed showing how the diverse hardware and software systems that must work together to create such an environment might be organised to support a university campus and a "smart lecture room" in particular. The VSA was applied to develop an architecture for the smart lecture room. One component, a smart camera, "SmartCam" was singled out for detailed design and software implementation. A summary of the SmartCam prototype follows.

The functional requirement of the Smart Camera is to detect if a person is present, that is, sitting in front of it, or not, and report the observation. Consistent with this objective, the Smart Camera is to operate as efficiently as possible. This requires the software of the Smart Camera to optimize resource usage given the dynamic nature of the environmental disturbances generated by the observed subject. Thus, the system is acting as an adaptive presence sensor. Based on these high-level requirements, the VSA methodology was applied to develop an architecture, a software design, an implementation, and finally, a test of the end product.

The class diagram for SmartCam is shown in **Figure 9**. As a first effort at building software based on the VSA architecture, the roles are mapped one-to-one to software classes. The SmartCam controller class is a container that instantiates the other classes. Notice the Plant has separate classes for image processing and accounting. An image processing component developed under another effort was used here. Also, it seemed reasonable to encapsulate the data storage and reporting in a separate class, hence the Accountant. A major design decision was made in order to get at the nature of the relationships between the components. Instead of using direct method calls on the class, interfaces all communications between components takes place via explicit message passing. The rationale behind this design decision is to explore the protocols that are required in the architecture. A benefit of this approach is that it makes distribution of the system almost trivial. This is a requirement for the Smart Environment implementation as the SmartCam controller may be located on a different machine than the camera and its local processing software. This need for distributed operation manifests itself in the SmartCamMSMQBus class. This class uses Microsoft's MSMQ messaging system to decouple the Plant from the controller. All communication between the various controller components and the Plant are transported by MSMQ. All messages in the system are expressed in the Extensible Mark-up Language (XML) and software tools are used to facilitate development using XML. There are twenty-three different message types.

Figure 10 shows the two main sequences of the SmartCam: routine operation and a state change. After a number of Plant reporting cycles, Operations sends a message to Planning requesting advice. Planning will evaluate its model of the Plant and issue a set of tuning parameters to Operations. This may result in Operations deciding to change some aspect of the Plant's processing. If so, an updated Plan is sent to the Regulator for further processing and transmission to the Plant. Other routine reports are shown. The second

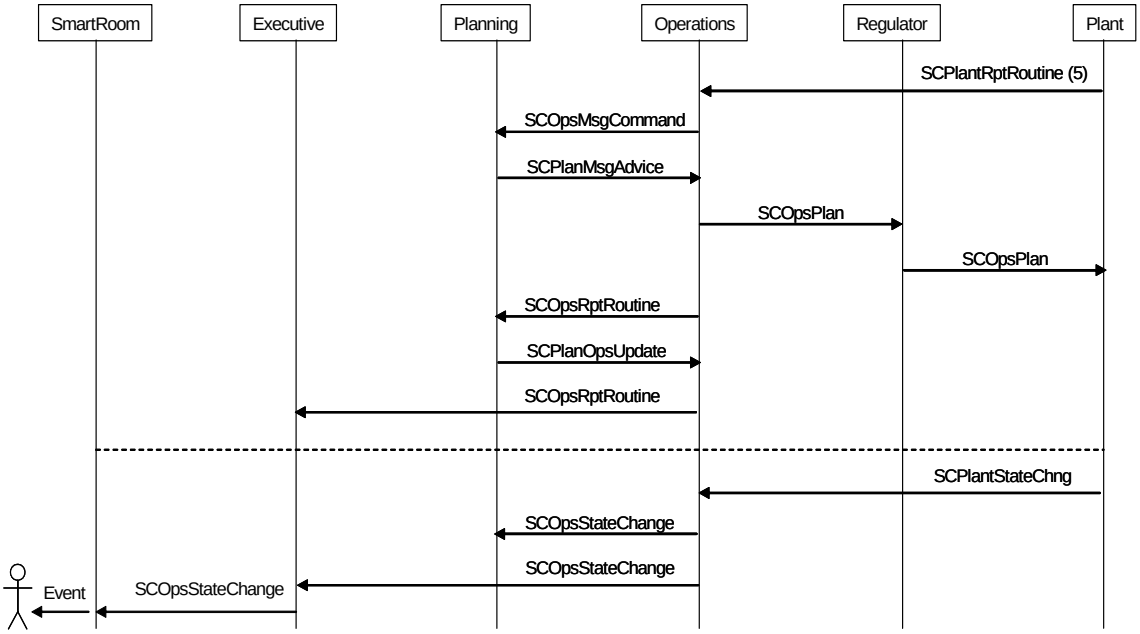

Figure 10 Routine Operation and State Change (observation)

sequence shows the effect of a State Change (presence state changes) and this triggers an evaluation cycle in Planning and Operations.

The behaviour of SmartCam as an adaptive controller is summarised as follows. The Plant has two critical control variables: PlantCycleTime and PlantReportsPerCycle. PlantCycleTime is the period (time interval in milliseconds) of observations and PlantReportsPerCycle is the number of cycles to include in a summary report sent to Operations. Operations processes the status reports from the Plant. After some number of reports (OperationCyclesPerReport) it then makes its own Cycle. Operations sends a message to Planning asking for advice on the Probability that the user will be present during the next time interval. Based on this probability it enters its Evaluation process. Information about the recent past state of the Plant and the Probability from Planning is used to calculate an ActivityLevel. ActivityLevel (Very Low, Low, None, High, Very High) is a fuzzy factor used to decide how to adjust the PlantCycleRate and the PlantCyclesPerReport. (The Plan is updated in the Regulator and retrieved by the Plant for execution.) During a cycle, Operations also sends a routine status report to both Planning and Executive. Based on these reports and its historical database, Planning advises Operations on adjusting its own cycle rates. Decision making based on information from Planning is an example of adaptive control. Planning must also evaluate and adjust its own cycle times with respect to the Plant and Operations. Each cycle and each report costs some number of System Accounting Units (SAU). The goal is to minimise this cost consistent with performance. As historical data about the behaviour pattern of the subject is gained, better overall system performance is provided by the adaptive control strategies. The controller is essentially tracking the subject.

A bar chart of the control variables (described above) of the SmartCam running in simulation mode is shown in **Figure 11**. In this run, Planning's historical database of observations and the Plant's simulation database are the same. This setting isolates the behaviour of adaptive algorithm. The period of the Plant's cycle is shown on the left scale in seconds. The number of Plant cycles per Operations cycle is shown on the *right* scale. The number of Plant cycles per report and the number of Operations cycles per report is plotted on left scale. The simulation database profiles someone who comes in (sits in front of the camera) with the following probabilities over time: 0900, .5; 0915 .6; 0930 .7; 0945 .9; and 1000 1.0. As seen in the figure, the period of the cycles increases as the probability (and the simulated observations) increases over time. Notice the large jump between Operations cycle 5 and 6 as the probability of the subject being present goes from 0.7 to 0.9.

5. Summary and Conclusion

The motivation behind the VSA approach is the need to design, build, and maintain a broad class of software systems that must operate in a dynamic environment and interact with humans (or their surrogates). Any software more complex than a simple input-process-output algorithm (an open loop controller) is not a Turing Machine [13]. If humans are involved, it is a complex adaptive system. Consequently, a major objective of this approach is to provide a path for the piece-meal introduction of human skills into software systems. To summarise, the overall goal of the VSA

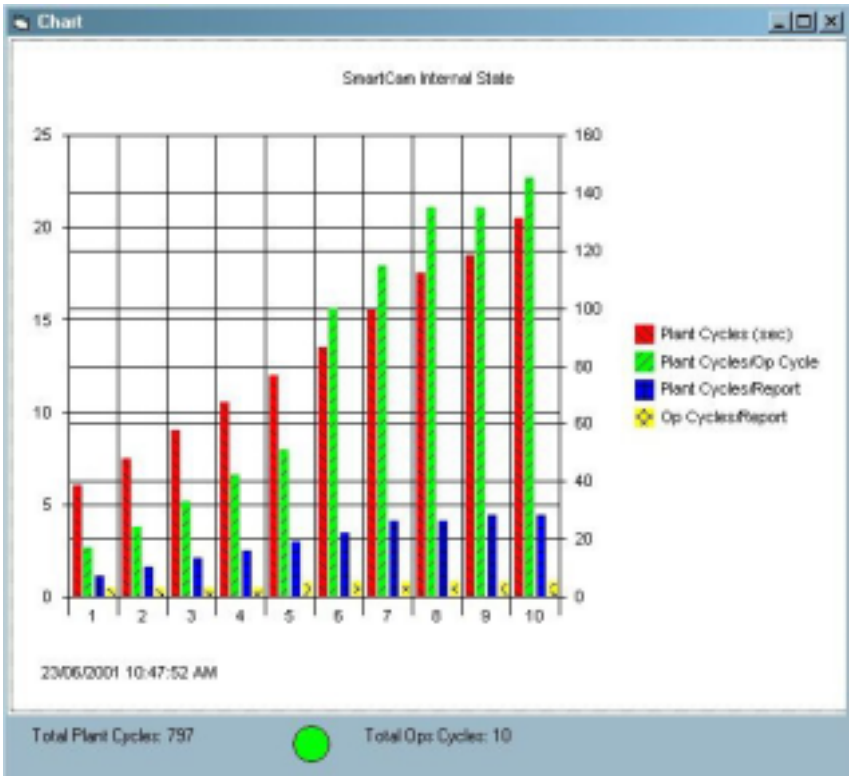

Figure 11 Snap-shot of SmartCam Running in Simulation Mode

approach is to provide a path to achieve *software viability*:

Software Viability is the quality a software system has if it can be adapted over time by humans (design-time adaptability) toward becoming an intelligent (supervisory-adaptive-control) system (run-time adaptability).

This statement is illustrated graphically in **Figure 12** with reference to examples of existing systems along a spectrum that approximates the stages of development leading to a viable software system. The Viable Software Architecture was presented in five stages and expressed in UML collaboration diagrams. The presentation unfolded the architecture, piecemeal in stages. These stages represent the major advanced capabilities a software application acquires along the road to viability. A cornerstone of this approach is that systems can be grown, over time, and with the benefit of feedback from each previous release toward becoming an intelligent control system. This approach recognises that such advanced systems cannot be designed "up front" or "top down." However, the hope is that with experience such systems can eventually be developed efficiently.

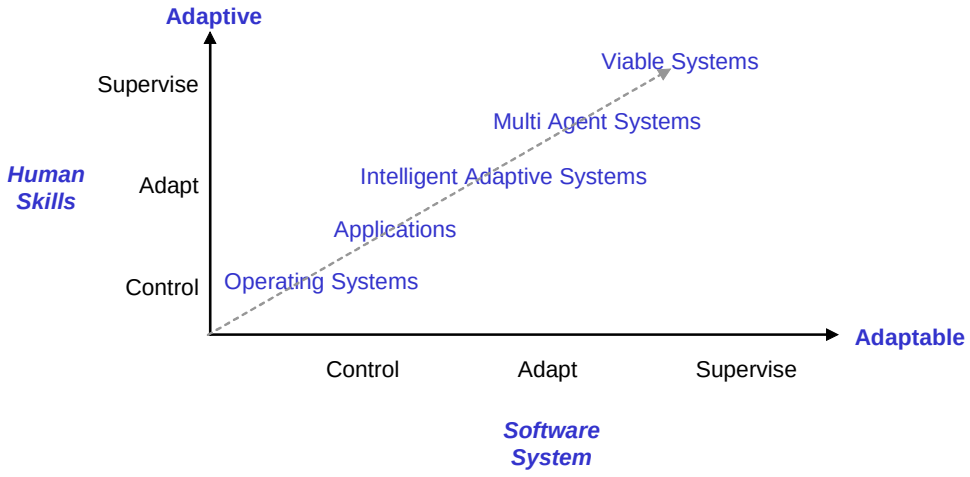

Figure 12 Migration of Human Skills into Software Systems

References

1. Beer, S., *Decision and Control*. 1956, Great Britain: John Wiley and Sons Ltd.
2. Beer, S., *Cybernetics and Management*. 1959, Oxford: English Universities Press.
3. Beer, S., *Cybernetics and Management*. Second ed. 1967, London: The English Universities Press LTD.
4. Beer, S., *The Heart of the Enterprise*. 1979, New York: John Wiley and Sons.
5. Beer, S., *Brain of the Firm*. 2nd ed. 1981, New York: John Wiley and Sons.
6. Beer, S., *The Viable System Model: Its Provenance, Development, Methodology and Pathology*. Journal of the Operational Research Society, 1984. **35**(1): p. 7-25.
7. Beer, S., *Diagnosing the System for Organizations*. 1985, Great Britain: John Wiley and Sons Ltd.
8. Beer, S., *The Evolution of a Cybernetic Management Process*, in *The Viable System Model: Interpretations and Applications of Stafford Beers' VSM*, R. Espejo and R. Harnden, Editors. 1989, John Wiley & Sons. p. 77-100.
9. Shaw, M., *Beyond objects: A software design paradigm based on process control.* ACM Software Engineering Notes, 1995. **20**(1).
10. Booch, G., *Object-Oriented Development.* IEEE Transactions on Software Engineering, 1986. **2**(12): p. 211-221.
11. Shaw, M. and D. Garlan, *Software architecture: perspectives on an emerging discipline*. 1996, Upper Saddle River, N.J.: Prentice Hall.
12. Herring, C., *Viable Software: The Intelligent Control Paradigm for Adaptable and Adaptive Architecture*, Dissertation, *Department of Computer Science and Electrical Engineering*. 2001, University of Queensland: Brisbane. p. 413.
13. Wegner, P., *Interactive Foundations of Object-Based Programming*, in *IEEE Computer*. 1996. p. 70-72.