Second International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008),
at the International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 2008, Pages 23-26

Visibility of Control in Adaptive Systems

Hausi Müller
Dep. of Computer Science
University of Victoria
Victoria, BC V8W 3P6 Canada
hausi@cs.uvic.ca

Mauro Pezzè
Faculty of Informatics
University of Lugano
CH-6900 Lugano, Switzerland
mauro.pezze@unisi.ch

Mary Shaw
Inst. for Software Research
Carnegie Mellon University
Pittsburgh, PA, 15213 USA
mary.shaw@cs.cmu.edu

ABSTRACT

Adaptive systems respond to changes in their internal state or external environment with guidance from an underlying control system. ULS systems are particularly likely to require dynamic adaptation because of their decentralized control and the large number of independent stakeholders whose actions are integral to the system's behavior. Adaptation may take various forms, but the system structure will almost inevitably include one or more closed feedback loops. We argue that adaptability is a characteristic of a solution, not of a problem, and that the feedback loop governing control of adaptability should be explicit in design and analysis and either explicit or clearly traceable in implementation.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/ Specifications; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/ Program Verification—*Validation*; D.2.11 [**Software Engineering**]: Software ArchitecturesPatterns

General Terms

Design, Documentation, Languages, Management, Performance, Reliability, Standardization, Theory, Verification

Keywords

Self-adaptive systems, autonomic systems, ultra-large scale systems, software ecosystems, continuous evolution

1. INTRODUCTION

Ultra large scale (ULS) systems, by virtue of their many participants, lack of central control, and inability to have complete specifications, and usually cannot be specified precisely. Consequently they need capability to ensure that system operation remains within the envelope allowed by system policy. Architectures based on closed-loop feedback control offer appropriate capabilities to address this challenge. Despite recent attention to software-intensive systems that adapt to unexpected changes in their operating environment or in their own state, development methods for these systems do not yet provide sufficient explicit focus on the feedback loop (control and data) that almost inevitably controls the adaptation. Even though system designers acknowledge the elements of a control loop and often speak informally about feedback in their systems, alarmingly often the design documents do not show the feedback loops explicitly. Layers of abstractions intended to hide complexity frequently also hide the control loops. The resulting lack of visibility makes it easy to neglect the control aspects in design which is so critical for validation and verification. Even worse, the lack of an explicit method makes it easy to neglect critical aspects of the analysis of the control.

In software engineering venues such as the recent Dagstuhl Seminar on Software Engineering for Self-Adaptive Systems and ICSE workshop series on SEAMS, DEAS, and WOSS, researchers often discuss the closed loop of control in their system analysis, but the actual control loops are frequently not evident in their architectural descriptions [8]. A major exception is the IBM Architectural Blueprint for Autonomic Computing which features the control loop as the central architectural component (i.e., autonomic element) [6, 9]. In contrast, the feedback loops in other engineering disciplines (e.g., electrical, mechanical or systems engineering), constitute an inherent and central part of design methodologies [3]. Perrow argues that tight coupling between components contributes to system failure, and feedback loops create coupling, especially if they are not visible in the system [11]. The idea of increasing the visibility of control loops in software architectures and software methods is not new. Over a decade ago, Shaw compared a software design method based on process control to an object-oriented design method [12]. She introduced a new software organization paradigm based on control loops with an architecture that is dominated by feedback loops and their analysis rather than by the identification of discrete stateful objects. Unsurprisingly, the process control pattern described in that paper resembles an autonomic element. If the computing pioneers and programming language designers had been control engineers by training instead of mathematicians, modern programming paradigms might feature process control elements.

Truex et al. suggested in a thought-provoking CACM paper that the problem with evolving software systems may lie in an incorrect goal set that we all have accepted [13]. The assumption that "software systems should support organi-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ULSSIS'08, May 10–11, 2008, Leipzig, Germany.
Copyright 2008 ACM 978-1-60558-026-5/08/05 ...$5.00.

zational stability and structure, should be low maintenance, and should strive for high degrees of user acceptance" might be flawed. They suggest an alternate view that assumes "software systems should be under constant development, can never be fully specified, and are subject to constant adjustment and adaptation". Certainly ULS systems match this view. Several studies conducted in 2006 seem to confirm that this notion of continuous evolution is taking hold [1, 2, 10]. In particular, the highly regarded Carnegie Mellon Software Engineering Institute (SEI) study on ULS systems suggests that traditional top-down engineering approaches are insufficient to tackle the complexity and evolution problems inherent in decentralized, continually evolving software [10].

To be able to observe and possibly orchestrate the continuous evolution of software systems in a complex and changing environment, we need to push the monitoring of evolving systems to unprecedented levels and increase the visibility of the control loops driving continuous evolution.

2. PROBLEMS VS. SOLUTIONS

Following Jackson [7] we begin with a distinction between problems and solutions.

- Solutions require effects on the domains of the problem. These effects are usually specified as phenomena in the problem domain, not explicitly as adaptations.
- "Adaptiveness" is a property of solutions: the design of the solution incorporates mechanisms to instrument and monitor the solution and its environment and to change the system behavior in response to observed changes.
- Some problems can be solved with either adaptive or nonadaptive systems. For some problems, adaptive solutions may be especially appropriate, but some problems admit of both adaptive and non-adaptive solutions.

We propose, therefore, to use the adjective "adaptive" to refer to properties of solutions and to find other adjectives to describe problems that might (or might not) be solved adaptively. Attributes of problems that tend to suggest considering adaptive solutions include:

- Uncertainty in the environment, especially uncertainty that leads to substantial irregularity or other disruption; that may arise from external perturbations, rapid irregular change, or imprecise knowledge of the external state.
- Nondeterminism in the environment, especially of a sort that requires significantly different responses at different times.
- Requirements, especially extra-functional requirements, that can best be satisfied through regulation of complex, decentralized systems (as opposed to traditional, top-down engineering) especially if substantial tradeoffs arise among these requirements.
- Incomplete control of system components, for example because the system incorporates embedded mechanical components, the task involves continuing action, or humans are in the operating loop (they are only biddable, not fully controllable.)

Implementations of adaptive systems are often "reflective". Reflective systems maintain an explicit representation of their system state and can refer to that state in order to modify their behavior. This can be an appropriate way to implement the model of internal state that is required for a feedback loop. So reflective mechanisms are *often* appropriate for adaptation, but the presence of reflection does *not necessarily* mean that a system is adaptive.

Implementations of adaptive systems cover significant design spaces from continuous to discrete control systems and from closed to open systems. A closed system is specified completely in terms of its behaviour, given internal state and I/O interface. In contrast, the specification of an open system must be developed in tandem with a description of the real-world domain in which it operates. An open system interacts with this world via sensors and actuators; it is often safety-critical, and its specification can be a challenge. Fly-by-wire and engine control systems are two examples of open systems. ULS systems cover this entire space of control systems. In this paper we concentrate open systems.

3. MAKE FEEDBACK LOOPS EXPLICIT

Software engineers are trained to develop abstractions that hide complexity. Hiding the complexity of a feedback loop seems obvious and natural. It follows that feedback loops, which are the bread and butter of adaptive systems and hence software-intensive and ultra-large systems, should be made first class design elements. We feel that designers will reap significant benefits by raising the visibility of control loops in the design and implementation of adaptive systems. ULS's are particularly prone to require feedback due to a diverse and varying set of stakeholders with independent goals, decentralized control, and uncertainty in the operating environment leading to evolving requirements, nonuniformity, inconsistency, instability, and rich tradeoff spaces.

Feedback Architectures

The purpose of a simple feedback process control system, which includes a control loop with sensors and actuators, is to maintain specified properties of the outputs of the process at (or sufficiently close to) a given reference value called the set point. For physical processes in the real world, it is rarely possible to do this with a preset sequence of controls (i.e., an open loop system). Instead, it is usually necessary to monitor the process and change settings of the operating controls to keep the outputs in the desired range. The feedback loop allows the control system to adapt to varying circumstances. Figure 1 shows a simple feedback loop.

Shaw identified a software architecture style based on simple process control that addresses problems with the attributes identified in Section 2 [12].

Self-adaptive Architecture

In Figure 1 the controller embodies a static model. Figure 2 depicts a model-reference adaptive system (MRAS).

In a typical MRAS, the parameters of the controller are

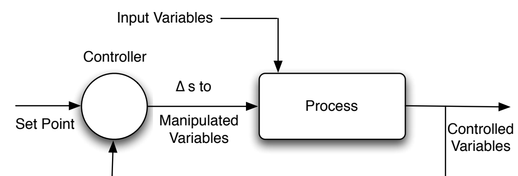

Figure 1: Simple process feedback control.

adjusted to satisfy the specifications of the reference model [4]. Feedback loops of this sort are used in almost all engineered devices to bring about desired behavior despite undesired disturbances.

At the 2007 Dagstuhl Seminar on Software Engineering for Self-Adaptive Systems, Shaw presented the feedback control architecture depicted in Figure 3. It is an MRAS that distinguishes between the executing process and its context or operating environment. Moreover, the sensors into the executing process and its context include sensors into current states and predictions about their future to reflect an accurate model of current and future state. Sensor events are filtered and stored in the model to remember the past and to predict the future. The controller compares goals of the feedback control system with the model and devises a plan of action to change the controller and the process. Representing the control explicitly in this form reveals obligations that fall on various activities of design and development:

In Requirements: Specify the goals in terms of functional and non-functional requirements (e.g., using goal models), including tolerances, trade-offs, time constants, stability and convergence conditions, hysteresis specifications, and context uncertainty.

In Design: Clearly identify all control and data elements of the adaptive system explicitly, preferably in a separate architectural control view. Choose an adaptation strategy to determine the model and plan to effect the adaptation. For example, Litoiu identifies four different kinds of performance models and corresponding implementation techniques [9]. Choose a monitoring strategy, including event formats, sampling rates (fixed/variable) and filters, to observe the state of the executing system and its context.

In Analysis/V&V: Validate how the current state is modeled from sensor data. Validate the correction plan. Show how corrections can be achieved with the available commands. Show that the corrections will have desired effects (i.e., global stability and convergence). Verify time constants.

In Implementation: Clearly map from elements of design to elements of implementation. Control is not necessarily a separate implementation component.

4. EXAMPLES

IBM developed an architectural blueprint for autonomic, self-managed systems [6]. The IBM architectural blueprint identifies the autonomic element as a fundamental building block for designing self-configuring, self-healing, self-protecting and self-optimizing systems. An autonomic element uses sensors and effectors, and is composed of a monitor, an analyzer, a planner and an executor that share a

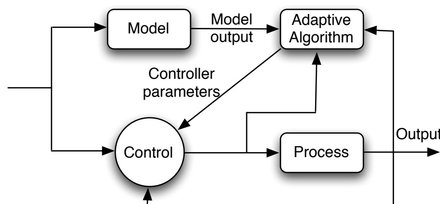

Figure 2: Model-reference adaptive control.

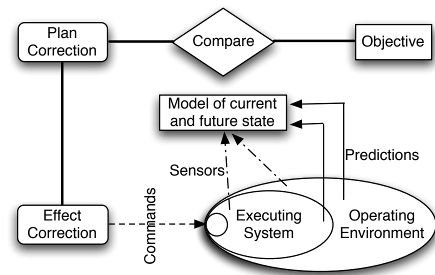

Figure 3: Feedback control for adaptive systems.

knowledge base. The goals for the control loop are specified through effectors (typically through parameters or policies). The monitor senses the managed process and its context, filters the accumulated sensor data, and stores relevant events in the knowledge base for future reference. The analyzer compares event data against patterns in the knowledge base to diagnose symptoms, and stores the symptoms for future reference in the knowledge base. The planner interprets the symptoms and devises a plan to execute the change in the managed process through the effectors. An interface consisting of a set of sensors and effectors is called a manageability endpoint. To facilitate collaboration among autonomic elements, the control and data of manageability interfaces is standardized across managed elements and autonomic building blocks.

Garlan et al. have developed a technique for using feedback for self-repair of systems [5]. Figure 4 shows their system. This mapping identifies the feedback loop in Garlan's system:

Model	**Example**
Executing system in its operating environment	Executing system with runtime manager
Sensors	Monitoring mechanisms
Predictions	(System is not predictive)
Objectives, model of current state	Architectural model
Compare	Analyzer
Plan correction	Repair handler
Effect Correction, commands	Translator, runtime manager

5. NEXT STEPS

Patterns for Adaptive Applications

Designers of software-intensive adaptive systems will already realize significant benefits by raising the visibility of control loops to first class and specifying the major components of the control loop explicitly (i.e., goal specification, model of past, current and future state, process and context monitoring and event filtering capabilities, analysis, reasoning and planning engines). Further benefits could be realized by identifying common forms of adaptation and then distilling design and V&V obligations for specific patterns.

As a first step, the adaptive control loops discussed above should be refined into a reference model, which in turn is reconciled with other reference models. Secondly, control theory has a long history with huge successes in all branches of engineering. Mining the rich experiences in these fields and applying them to software-intensive adaptive systems is

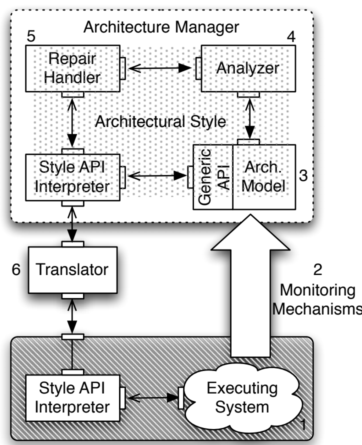

Figure 4: Self-repairing system with feedback.

a key step [4]. As a third step, we should identify design patterns that are used in practice, and codify them as patterns to make the structure and obligations explicit.

Multiple Control Loops

Systems often have multiple control loops. Good engineering practice calls for making them independent whenever possible. Sometimes they can operate independently, and sometimes they form a coherent hierarchy, but other types of interaction are possible.

When multiple control loops interact, system design and analysis must cover their interactions. As control grows more complex, it is especially important for the design and analysis to be explicit. For example, Litoiu [9] discusses hierarchical control in a class of quality of service and service oriented architecture applications.

Catalogs

Engineering practice relies on good reference material, so the classes of simple and multiple control loops should be cataloged systematically. Identifying common forms of control loops and defining patterns are still largely open research topics. Here, we draft few illustrative examples.

Control loop patterns depend on adaptation goals. For example, a class of self-optimizing control loops can be devised by expressing obligations as performance thresholds to be met at system level, and by defining mechanisms to dynamically collect performance data at component execution level, instantiate models to relate performance data at component level with system level obligations, and develop relocation mechanisms to reduce component execution time to avoid violations of system level obligations. V&V obligations are based on proofs that the relocation mechanisms can reduce execution time of the critical components without interfering with the performance of other components, and that improving component performances impact positively on the overall performance obligations. Such control loops may be applied at different abstraction levels, but would always require the availability of performance data at the lower level and the possibility of dynamic component relocation to enact adaptation strategies.

Self-healing control loops can be devised by expressing obligations as assertions at user level, and by defining mechanisms to map assertion from system to component level, identify elements that affect assertion validity, define mechanisms that eliminate components that cause assertion violations from the execution. V&V obligation may compute the impact of component workaround on system functionality.

Acknowledgments

This work grew out of discussions at Dagstuhl Seminar 08031 on Software Engineering for Self-Adaptive Systems (13-18 January 2008). We thank the Dagstuhl participants, especially Luciano Baresi, Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, and Andreas Rasche, for stimulating discussions that led to the insights reported here. This work was funded in part by the National Science Foundation (ITR-0325273) via the EUSES Consortium and under Grants CCF-0438929 and CNS-0613823 and National Sciences and Engineering Research Council (NSERC) of Canada (CRDPJ 320529-04) and IBM Corporation via the CSER Consortium and the Swiss National Foundation via the PerSeOs project.

6. REFERENCES

[1] B. Boehm. A view of 20th and 21st century software engineering. In *Proceeding of the 28th International Conference on Software Engineering*, 2006.

[2] M. Broy, M. Jarke, M. Nagl, and D. Rombach. Manifest: Strategische Bedeutung des Software Engineering in Deutschland. *Informatik-Spektrum*, 29(3):210–221, 2006.

[3] R. S. Burns. *Advanced control engineering.* Butterworth-Heinemann, 2001.

[4] G. A. Dumont and M. Huzmezan. Concepts, methods and techniques in adaptive control. In *Proceedings of the 2002 American Control Conference*, 2002.

[5] D. Garlan, S. Cheng, and B. Schmerl. Increasing system dependability through architecture-based self-repair. *Architecting Dependable Systems*, 2003.

[6] IBM. An architectural blueprint for autonomic computing, June 2005.

[7] M. Jackson. *Problem frames:analyzing and structuring software development problems.* Addison-Wesley, 2001.

[8] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering*, 2007.

[9] M. Litoiu, M. Woodside, and T. Zheng. Hierarchical model-based autonomic control of software systems. *SIGSOFT Software Engineering Notes*, 30(4), 2005.

[10] L. Northrop, P. Feiler, R. P. Gabriel, and et al. Ultra-Large-Scale systems - the software challenge of the future. Technical report, Software Engineering Institute, Carnegie Mellon, June 2006.

[11] C. Perrow. *Normal Accidents: Living with High-Risk Technologies.* Princeton Univ. Press, September 1999.

[12] M. Shaw. Beyond objects: a software design paradigm based on process control. *SIGSOFT Software Engineering Notes*, 20(1):27–38, 1995.

[13] D. P. Truex, R. Baskerville, and H. Klein. Growing systems in emergent organizations. *Communications of the ACM*, 42(8):117–123, 1999.