Control Theory-Based Foundations of Self-Controlling Software

Mieczyslaw M. Kokar, Kenneth Baclawski, and Yonet A. Eracar, Northeastern University

THE AUTHORS' CONTROL THEORY-BASED PARADIGM GIVES A FRAMEWORK FOR SPECIFYING AND DESIGNING SOFTWARE THAT CONTROLS ITSELF AS IT OPERATES. BASED ON THIS PARADIGM, THEIR SELF-CONTROLLING SOFTWARE MODEL SUPPORTS THREE LEVELS OF CONTROL: FEEDBACK, ADAPTATION, AND RECONFIGURATION.

ALGORITHMS WITH EMBEDDED control and adaptation are common in software systems. Some examples are

- software for dynamically adjusting a database-management system's buffering strategy,
- routing algorithms for networks,
- load-balancing algorithms for distributed computer systems,
- graphical user interfaces that adapt to the user, and
- caching strategies for OS memory management.

All these software subsystems interact with an environment that could be the external physical world or another layer of the computer system. Such environments can be characterized as *dynamic systems*.[1] The essence of a dynamic system is that its output depends on the system's state. So, the system does not shift dramatically from one output to another (in response to changes in the input) but exhibits some form of inertia (because of the dependence on state). When designing a software system that interacts with a dynamic environment, software engineers need to take into account the dynamic characteristics of the environment and computer system, or the system might behave differently from what the engineer expected.

In computer systems, the first observed form of inertia was *locality of reference*. This property has been exploited heavily ever since in both hardware and software. Hardware devices use caches and buffers to exploit locality of reference. Operating systems use locality of reference to improve the performance of virtual memory and file systems. Communication networks have two kinds of locality of reference. *Physical* locality of reference is the tendency for communication to be between computers that are physically near one another. *Temporal* locality of reference is the tendency, once a pair of computers has communicated, for that pair to communicate again in the near future, and then repeatedly.

In spite of the obvious analogy of software systems to control systems, the basic paradigm of control has not found its place as a first-class concept in software engineering. For instance, Mary Shaw and David Garlan use the control architecture to identify an architectural style that they call the *process-control paradigm*.[2] However, they consider the possibility that only the controller that controls a physical system (in control terminology, such a controlled system is called a *plant*) is implemented in software. They do not consider a plant that is itself software. Also, they do not go beyond the basic control model.

Control theory generally concerns systems that repeatedly interact with the world through a sense-response-act loop. Applications that can exploit this pattern are common in software engineering. Two examples are the read-evaluate-print loop of traditional batch processing or the event-dispatch-handle loop of graphical user interfaces. These examples' common features have been abstracted in the controller object-oriented design pattern.[3] The controller design pattern does not have an explicit feedback mechanism, so it represents only the simplest form of control model—namely, the *open-loop model*, which we'll describe in the next section. The control-theory paradigm goes beyond open-loop systems; it includes, for example, the read-match-fire loop of rule-based systems and the sense-response-act loop of intelligent agent systems.

Significant advances can be achieved by mapping the concepts of control theory to software engineering and then transferring the concepts and tools developed in control theory—for example, controllability, stability, and sensitivity analysis. Toward that end, we propose a new paradigm for software development that explicitly and systematically addresses self-control of software. This paradigm

- regards the software system as a plant to be controlled;
- models the behavior of the plant and the environment as a dynamic system;
- identifies measurable inputs to the plant and classifies them as *control inputs*, which control the plant's behavior, or *disturbances*, which alter the plant's behavior unpredictably;
- includes a *controller* subsystem for changing the values of the control inputs to the plant; and
- adds, if necessary, a *quality of service* (QoS) subsystem for computing feedback. The controller uses this feedback to control the plant.

This paradigm can exploit the considerable research and industrial experience in control theory. Also, with this paradigm we can systematically derive models for self-controlling software.

Control theory-based software models

A self-controlling software system distinguishes two primary entities: the computer system and its environment, also called the *world*. Typically, the world is a dynamic system whose behavior is a function of its previous state, actions exerted on it by the computer system, and time. We presume that the computer system attempts to satisfy an externally defined goal through the appropriate selection of actions. Actions are generated based on various sensory inputs, the goal, and the computer system's internal state. Figure 1 illustrates a basic software system.

We now introduce a series of progressively more complex control problems and describe the control models that the control community has developed to address these problems. These models add redundancy to the basic function (plant) of a software system to introduce various levels of self-controllability to the overall system.

An image-recognition example. Throughout the article, we'll use a hypothetical image-recognition system[4,5] to illustrate our ideas. The system's basic function (the part that does not involve any control) is simply to recognize certain features of input images. For simplicity, assume that only two classes of inputs are possible: *square* and *nonsquare*. The program's goal is then to classify input images into these two classes.

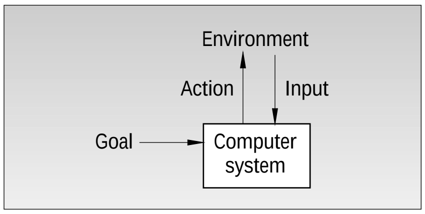

Figure 1. The basic system model.

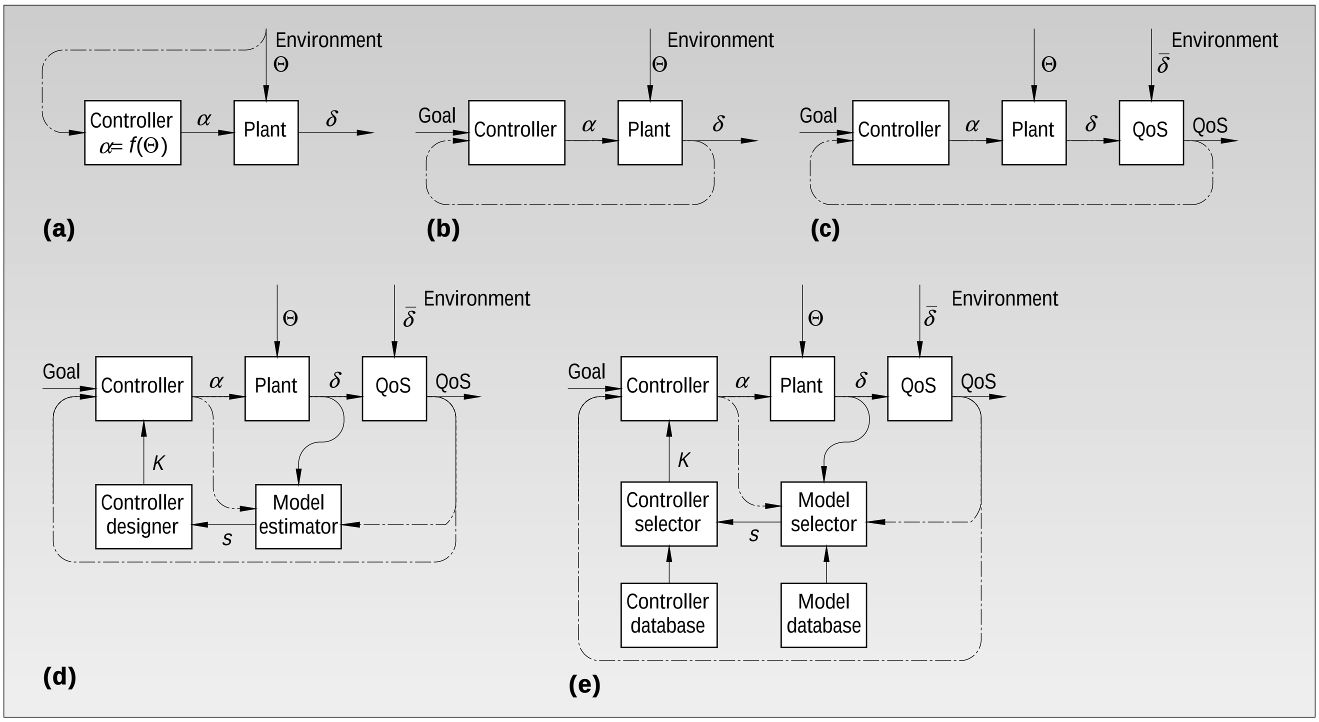

Figure 2. Control models: (a) open-loop; (b) closed-loop (feedback); (c) closed-loop with a quality-of-service subsystem; (d) indirect-adaptive; (e) reconfigurable.

Suppose that the program consists of two components: edge identification and object classification. Let's focus on object classification. Assume this component is implemented as a statistical classifier that tests the hypothesis that the image is a square based on a confidence coefficient α. It outputs its classification decision δ. The inputs to this component are edges represented as classes Θ_i of gradient values for each edge point. The means $\overline{\Theta}_i$ and variances s_i for these classes are also computed by the plant and used in its hypothesis testing. The edge-detection component detects the edge points and calculates the classes Θ_i. When no controller is involved, the value of α is selected at the time of system design and remains fixed throughout the classifier's life.

Open-loop control. This model selects the control-input value according to a *control law* that calculates the control input based on the values of other inputs. The control law is part of the controller. Unlike the basic system (see Figure 1), the open-loop control model splits the system into a plant and controller.

In the image-recognition example, this model could involve the calculation of the confidence coefficient α as a function of the inputs, $\alpha = f(\Theta)$. Specifically, this function could be defined such that for inputs with higher variances, the confidence coefficient α would have a lower value.

Figure 2a shows the information flow in this example. In this and later figures, the annotations on the arrows illustrate how the model would be used for the image-recognition example. Arrows with no source are inputs from the environment; arrows with no target are outputs to the environment.

Closed-loop (feedback) control. This model explicitly and immediately feeds back the plant's output to the controller (see Figure 2b).

Although control theory assumes that the plant directly provides feedback, we can't assume this for software systems. So usually, a QoS subsystem must be introduced. In some cases, it computes the feedback's value as a function of input and output variables. In other cases, it might take an external input (for example, from the user) and then compute the feedback's value based on all available information. The controller uses the feedback produced by the QoS subsystem to compute control inputs. Figure 2c shows the feedback model with a QoS subsystem.

This control model is more precisely expressed as a *feedback loop*. In each loop iteration, the plant receives input from the environment, makes a decision based on the input, and acts by affecting the environment and then producing output signals. It sends these signals to the QoS subsystem and then to the controller. To make the decision, the plant uses input from the controller and internal state information as well as the sensor input. The controller receives plant output through the feedback loop. It also knows the control goal. It then evaluates whether the goal is satisfactorily being achieved. If not, the controller changes its input to the plant to achieve the control goal. (In software engineering, this model is often classified as an adaptive software model, but it would not be classified as adaptive in control theory.)

In our image-classification example (see Figure 2c), the feedback is the probability of correct recognition, *PCR*(*t*), for each input frame *t*. This probability is updated by the following equation, which is part of the QoS subsystem:

$$PCR(t) = \frac{PCR(t-1)\cdot(t-1)+\overline{\delta}}{t},$$

where $PCR(0) = 1$ and $\overline{\delta}$ is the external feedback input (in our example, the correct classification decision). The goal is to achieve the highest probability of correct recognition—that is, $PCR(t) = 1$. The control input α is computed by

$$\alpha(t) = K \cdot (PCR(t-1) - PCR(t)) + \alpha(t-1).$$

Adaptive control. Many of today's control problems make high demands on the controller. These problems inherently involve large-range dynamic disturbances of all kinds and stringent time requirements. Also, the disturbances occur and change unpredictably, causing an unpredictable response because of the plant's nonlinear characteristics. Changing the goal can add yet another range of disturbances.

To solve this problem, Karl Åström introduced *adaptive control*,[6] which can deal with the uncertainty in the model parameters of the controlled plant. The exact values of these parameters and of the controller do not need to be known when an adaptive controller is designed.

There are two general approaches to adaptive control. *Direct adaptive control* parameterizes the plant model in terms of the controller parameters. It then estimates the controller parameters directly. *Indirect adaptive control* estimates plant parameters online and uses them to calculate the controller parameters. This approach adds two subsystems: the *model estimator* and *controller designer*. It adjusts the controller parameters based on the plant's model that is being updated during execution. Figure 2d shows the software model for indirect adaptive control.

For our image-classification example, the plant is a statistical classifier, so the plant's model is probabilistic and is represented by a normal distribution $N(D_i; \mu_i, \sigma_i)$, where D_i is the angle between two consecutive edges Θ_i and Θ_{i-1},

$$D_i = \overline{\Theta_i} - \overline{\Theta_{i-1}}.$$

The model estimator updates the model (the estimates for μ_i and σ_i) incrementally after receiving input from the plant (the mean edge gradient $\overline{\Theta}_i$ and variance s_i^2) according to this rule:

$$s(D_i) = \sqrt{\frac{s_i^2}{n_i} + \frac{s_{i-1}^2}{n_{i-1}}},$$

where n_i is the number of edge classes for image *t*. The model estimator averages the updated estimates for σ_i over all pairs of consecutive edges and passes the estimates σ to the controller designer. The controller designer updates the gain *K* of the control law according to $K = C \cdot \sigma$, where $\sigma = s^2$ and *C* is a fixed constant.

Reconfigurable control. Adaptive control, although more flexible than conventional feedback control, has its own limitations. The most obvious limitation is that the logic for both the identification and decision functions is implemented at the time of controller design and remains fixed for its lifetime. Thus, the adaptive controller has a limited ability to update the control law: it can only update the control-law parameters within a predefined class of models (the parametric uncertainties of the model). It cannot, however, deal with all kinds of nonparametric uncertainties, including high-frequency unmodeled dynamics, low-frequency unmodeled dynamics, and sensor noise. Sometimes when the plant has a characteristic even slightly different than that presumed in the adaptive-controller design, the results can be catastrophic. As Charles Rohrs and his colleagues showed, when a plant's dynamics are not modeled correctly, even small uncertainties can lead to severe parameter drifting and plant instability.[7] Much

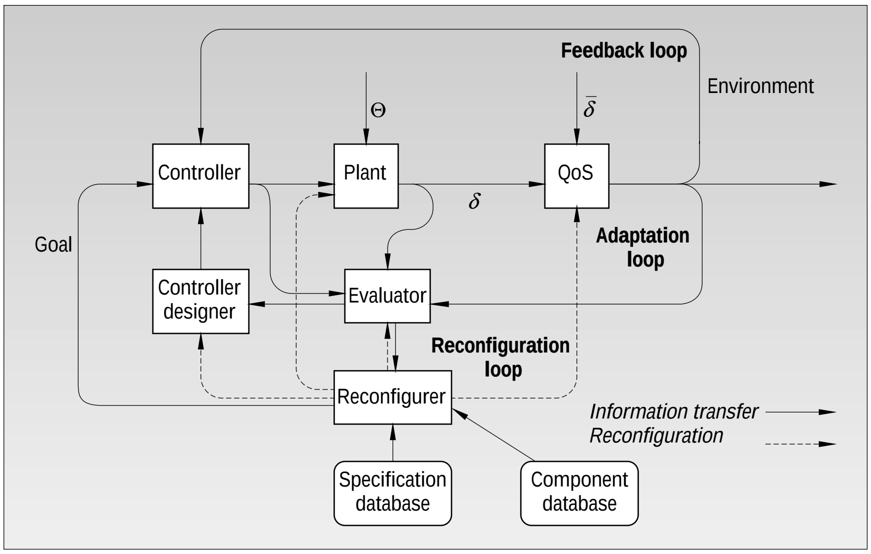

Figure 3. The self-controlling software model combines and generalizes the features of the control models shown in Figure 2.

research has been performed to address the basic problems that Rohrs and his colleagues pointed out.

Moreover, even if an adaptive controller can adapt to new situations, a nonlinear characteristic of the controlled plant might periodically resurface. Adapting to this recurring situation every time seems unreasonable. A much more economical approach would probably be to learn a control law associated with a particular dynamic characteristic type, store it in the controller's database, and use it whenever the recurrence of a known situation is recognized. This requires an intelligent controller with both adaptive and learning capabilities that not only can adapt to and memorize the new control law, but also can select an appropriate control law.

Improving upon adaptive control. Reconfigurable control[8] is a relatively new model in the design and implementation of control systems. The driving force behind this approach's development was the need to control plants that unpredictably change their dynamics structurally. This means that at different points in time, the plant's dynamic model must be described by equations having different variables and different mathematical operators. The main idea is to be able to monitor the situation, recognize structural changes, and then redesign the controller in real time to compensate for the structural changes.

Figure 2e shows the software model based on a reconfigurable controller. This model contains two new subsystems—the *model selector* and *controller selector*—and databases—the *model database* and *controller database*. The model selector incorporates all the features of the model estimator in the adaptive control model. Additionally, when it detects significant changes in the model, it can select a different model from its model database. This triggers the selection of a new controller from the controller database.

Reconfigurable control is applicable in many situations, such as damage to the plant. Damage need not result in a catastrophe. In many cases, radically changing the control strategy can compensate for damage. This is possible when redundancy exists in the controlled system. For instance, imagine a two-legged robot, whose right leg has been damaged, using the left leg for moving (jumping), like humans or animals would naturally do.

Referring to our image-recognition example, the model selector might decide that the normal distribution model is inappropriate for a given sequence of input images. Its database might have other probabilistic models, represented, for example, by the Poisson distribution, Weibull distribution, or gamma distribution. Selecting one of these models requires selecting a new controller and the controller designer's rules for updating control laws.

Reconfiguring the plant. In control, the plant is a physical object whose basic structure remains fixed over the control system's lifetime. Changing a plant would require redesigning its mechanical and electrical parts, manufacturing, installation, and so on. However, we're dealing with plants that are software systems. Because of software's great flexibility, we should be able to more easily reconfigure a plant. In this process, we are guided by such constraints as inputs from the environment and control goals. We are free to change the plant's algorithms for as long as this guarantees the achievement of the control goals.

For our image-recognition example, consider the edge-detection component. The performance of edge-detection algorithms depends on various characteristics of the input images. For instance, we can have two algorithms for edge detection: a *Sobel edge detector* and a *Laplacian edge detector*. The former works better with horizontal and vertical edges, while the latter is better for finding edges that are not perpendicular to the axis and that are not straight lines. A software system based on the reconfigurable-control model will select an edge detector appropriate to the specific type of input.

The self-controlling software model

Our *self-controlling software model* (see Figure 3) combines and generalizes the features of the control models we just described. This model's structure is basically three loops, each of which represents a different timescale for control activity:

- In the *feedback loop*, the controller sets parameters for the plant based on the goal and feedback received from the QoS subsystem.
- In the *adaptation loop*, the *evaluator* evaluates the behavior and performance to determine whether the plant's model is appropriate. It then adapts the model, if necessary, which in turn triggers a change in the control law.
- The *reconfiguration loop* is a drastic and relatively costly action that the *reconfigurer* performs at the evaluator's request. The reconfiguration can involve structural changes in the plant model, QoS subsystem, evaluator, controller, controller designer, goal, or even plant. The reconfigurer itself remains fixed. During decision making, the reconfigurer uses the *specification database*, which contains a high-level system requirement, including a high-level goal. During reconfiguration planning, it uses the *component database* to assemble various system elements.

We believe this model can lead to software systems with an impressive capability for responding, adapting, and reconfiguring. Of course, self-controllability does not come for free. As we mentioned before, the applica-

SOFTWARE AD GOES HERE. DALE TO PROVIDE TEXT, TONI TO CREATE.

tion's functionality must be supplemented with some redundancy to implement the mechanisms of self-adaptability: evaluation, model estimation, adaptation, and reconfiguration. However, we can reduce this overhead and improve overall system performance by

- evaluating the behavior based on a sample of feedback iterations rather than on every iteration;
- generating more efficient interfaces between components at runtime; and
- constructing more efficient component organizations, scheduling algorithms, and evaluation algorithms at runtime.

In the following subsections, we give a more detailed definition of the self-controlling software model by discussing the responsibilities of the various subsystems and the approach we have taken to realizing these subsystems in our research prototype systems.

The feedback loop. This loop consists of the controller, the plant, and the QoS module. The controller must adjust the plant's parameters to maintain the quality of service. In adjusting these parameters, the controller must obey various constraints. The most important constraints from the control theory point of view are the controllability and stability constraints. The controller must be designed such that the whole system is stable; that is, small changes in the control input do not cause large changes in the system's behavior. The system must also be controllable; that is, the controller should be able to drive the system to achieve its goal. To design a controllable and stable system, the controller's designer must know the plant's dynamic characteristics. (For other constraints, see the sidebar.)

The plant's dynamic characteristics depend on the software it executes and the hardware on which it executes. But the plant's main role is to perform some computation for the environment. Our approach treats the environment as an external system, its impact on the plant as disturbances, and its impact on the model as perturbations. The method we use to model the environment depends on its type. For instance, if the environment is constant or steady-state, the modeling job is much simpler than when the environment has a significant dynamic component.

In some situations, we might not need to model the environment (the disturbances). Nevertheless, we still need a good model of the plant's dynamics. In principal, we can derive such a model from models of the software and hardware, such as state-chart models, Petri net models, or logical models expressed in a temporal logic. In this case, the main problem becomes optimization—that is, optimizing the plant's performance with respect to the performance-evaluation measure developed by the evaluator.

When the environment is dynamic, we need to know its model to design the controller. Environments are either *continuous* or *discrete*. For continuous environments, the models are given by differential equations; for discrete environments, they are given by difference equations. Controllers can be classified along the same lines. When a controller is implemented in software, it will not necessarily be continuous, and might be either discrete or *hybrid*. In hybrid control, both the plant and the controller have continuous and discrete dynamics and variables.[9]

Dynamic environments are the center of attention of the whole control research community. We believe that when dealing with this kind of environment, the software engineer developing a self-controlling software system should seek a control engineer's expertise, to exploit his or her knowledge of controller design.

Evaluation of the plant's behavior and performance is based on quality of service.[10] The QoS module computes a measure of behavior and performance that takes the form of a multidimensional function similar to the *benefit function*.[10] The difference is that the benefit function measures the benefits received by an end user, while the QoS module measures how well the plant and its components perform relative to the system's specified mission. The evaluator measures the system's performance using either a quantitative measure of how well the system performed or a probability of correct operation for specific missions.

The adaptation loop. This loop adds two components to the feedback loop: the evaluator and the controller designer. If the self-controlling software includes only the feedback loop, the plant's model is not represented explicitly in the system; the human designer uses it to derive a control law (to design the controller). When the adaptation loop is implemented, the evaluator knows that model. The evaluator generalizes the concept of the model estimator in Figure 2d and model selector in Figure 2e. In the adaptation loop, the evaluator assesses whether the plant's behavior is compatible with the plant's model and compensates the model's parameters accordingly. The controller designer uses these model parameters to update the control law. It then passes the updated control law to the controller, which uses the control law to compute new conrol inputs to the plant.

Similarly to the feedback loop, self-controlling software that implements the adaptation loop must satisfy the additional constraints for an adaptive controller. Again, the most important constraints are controllability and stability.

The reconfiguration loop. This loop extends the adaptive model by adding three components: the reconfigurer, the specification database, and the component database. Additionally, the evaluator's role is extended by including the ability to evaluate the whole system's performance relative to the environment's variability. The component and specification databases store replacement components (and their specifications) that can be reconfigured: plants, QoS modules, controllers, controller designers, and evaluators.

A self-controlling system's ability to adapt to the environment's variability can be measured with the *Total Requirements Volatility* measure.[11] Strictly speaking, the TRV measures the volatility of requirements, not that of the environment. However, these two are closely related. Functional requirements can be expressed in many ways, but one of the most common is through preconditions and postconditions. Preconditions on the input parameters of a function represent the ranges or other constraints on input parameters that are required for the function to perform adequately. The environment's volatility is manifested in input values that do not satisfy the preconditions. To handle such input values, we must modify the preconditions—that is, modify the requirements. Such modifications are precisely what the TRV measures.

We considered using *function points* instead of the TRV for measuring requirements volatility. Although function points are more popular among software engineers, they are not appropriate for this problem. Unlike the TRV measure, the properties that define the function-point measure do not reflect changes in the environment.

Reconfiguration can be accomplished by *component selection*, *transformation*, and *composition*. In the first and simplest case, when the component does not perform according to the criteria set forth by the evaluator, the reconfigurer searches the component database for a replacement. Toward this goal, it first matches the interfaces of the components. If it finds a matching component, it checks the component's specification. We intend to use formal specifications for defining the purposes and interfaces of components. An algebraic specification of a component contains sorts, operations, and axioms. Collectively they define the component's function. To compose various component specifications, we must also specify the interfaces among the components (when the components are independent processes). A component can replace another component if its specification satisfies that of the component being replaced. This relatively simple operation might not be successful for a given specification database. Component transformation and composition can then be invoked to achieve the goal.

Researchers are investigating various approaches to transformations and composition for software architecture.[12] For instance, SRI is developing a provably correct approach to

Issues in the use of control theory for software engineering

The following issues, which have been investigated in control theory, are relevant to software engineering:

- *Controllability*: This is the ability to steer the system (plant) in desired directions. This concept is well-defined in control theory. A plant must be proved controllable to ensure that it will perform according to the specifications when inputs to the plant change in unexpected (but bounded) ways. Control theory offers good tools for analyzing the controllability of linear systems. Tools for analyzing the controllability of software in a more general sense need to be investigated.
- *Observability*: This is the ability to determine a system's (initial) state from measurements of the system. To control the system's state, we must be able to determine its current state.
- *Stability*: Basically, this is the system property that ensures that small changes (disturbances) in an initial state (also called an *equilibrium state* or *invariant set*) eventually have negligible effects on the system's behavior. It might mean, for instance, that the system does not exhibit oscillations as a result of small input. Control theory provides many (two dozen or so) definitions of stability.[1] Analyzing these definitions and assessing their appropriateness are important for the analysis of software stability.
- *Robustness*: This property is the controller's ability to achieve its objectives even if large, unanticipated variations occur in the plant. In other words, as the environment departs from the domain for which the system was designed, the system degrades gradually in performance rather than exhibiting a catastrophic failure.
- *Autonomy*: Because a self-controlling system performs reconfiguration without direct supervision, it exhibits a great deal of autonomy. Autonomy is intelligent control's focus.
- *Generality*: A system's generality is limited by its knowledge base. When it encounters a situation that the knowledge base does not cover, even its general-purpose reasoning mechanisms will fail. So, to be able to adjust to new situations, a self-controlling system should be able to incorporate new knowledge. The self-controlling software model is amenable to any number of control strategies, such as expert control, neural and fuzzy control, hybrid control, and learning control. All these methods (used in intelligent control) deal with both autonomy and generality.
- *Chattering*: If the environment reaches a state that is on the boundary between two control regimes, the system might chatter—that is, reconfigure repeatedly between two or more configurations. Because reconfiguration incurs an overhead, chattering can degrade overall system performance.

In addition to the control-theory issues, research on the self-controlling software model must address these software-system issues:

- *Scheduling*: Component scheduling is important for any system model. It is especially important for a system using the self-controlling software model because activities take place on multiple loops implemented on different time scales.
- *Proactive reconfiguration*: Reconfiguration is normally *reactive*; that is, it's triggered by an evaluation indicating that the system is not accomplishing its mission. The software model should also provide for proactive reconfiguration—a speculative reconfiguration that occurs because the system has determined that another configuration will likely perform better.
- *Efficiency*: The model should minimize the cost of the redundancy necessary for self-adaptability. For instance, in optimal control, the controller's goal is to minimize a cost function.

Reference

1. K. Passino and K. Burgess, *Stability Analysis of Discrete Event Systems*, John Wiley & Sons, New York, 1998.

the hierarchical refinement of software architectures. This approach first specifies an architecture in SADL, an architecture specification language, and then uses several provably correct transformations to progressively refine the architecture. Logical theories represent specifications. Refinement is guided by *patterns* and *styles*, which are structure-preserving mappings on theories. To address component composition, this approach uses *dynamic architectures*, which reconfigure the software architecture while the program is running.[13] The SRI approach addresses such problems as how to select components, connectors, and topologies (interface matching) and how to decide whether a particular architectural solution satisfies given architectural constraints.

Our self-controlling software model adds to this transformation model these elements:

- architectural components, such as the plant and controller, that are well-defined in control theory;
- an architecture that has evolved as a result of many years of research by the control community; and
- a set of control-theoretic constraints, such as controllability and stability, which have been proven useful and adequate.

SOFTWARE SYSTEMS REPRESENT some of the most complex artifacts ever created. Yet, as software's name suggests, it is not embodied in immediately tangible physical structures but in the electronic memories of computers. In contrast, the term hardware generically refers to the tangible physical artifacts associated with complex artifacts. In theory, software should be more flexible than hardware because it can be manipulated at electronic speeds without altering physical structures. Yet reconfiguring software to achieve its purpose better is often much more difficult than reconfiguring hardware. Even upgrading a software component from one version to the next one can be very difficult and has caused major system failures. Peter Neumann's ACM Risks Forum mentions several examples of these.[14]

Recent research in dynamic architectures has addressed the problem of automatic reconfiguration of software at runtime. But this recent research has not addressed the problems that occur when the environment is dynamic and the reconfigured algorithms produce unexpected consequences. For instance, the system might not properly steer toward the desired goal (the controllability problem), and uncontrolled oscillations might occur in response to small changes in the input (stability problems).

Control engineers have long observed these kinds of problems, and control theory has developed a host of concepts, architectures, and techniques to deal with these problems. These techniques include functional elements such as the controller, the model estimator, and the control designer. In addition, many classes of plants have been analyzed, and architectures for organizing the functional elements have evolved. Control engineers have identified important properties that self-controlling systems should satisfy, such as controllability, stability, observability, and robustness. Mathematical techniques and heuristics for designing and analyzing control systems have been developed.

Software systems are increasingly important components of the world's social and economic infrastructure. The expectations for availability, performance, and reliability are continually rising, which has led to the need for self-controlling systems capable of online reconfiguration. Such systems have already been developed in research labs and will soon be applied in industrial and commercial settings. Although mapping control theory concepts to software engineering is not easy, we believe that these concepts can make an important contribution to the development of these and other large, complex software systems. An architecture such as we've described in this article will expedite this mapping, letting software engineers exploit the vast amounts of knowledge and experience accumulated in control theory.

References

1. M.D. Mesarovic and Y. Takahara, *Abstract Systems Theory*, Springer-Verlag, Berlin, 1989.
2. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, Upper Saddle River, N.J., 1996.
3. C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Prentice-Hall, 1998.
4. Y.A. Eracar and M.M. Kokar, "An Architecture for Software That Adapts to Changes in Requirements," to be published in *J. Systems and Software*, 1999.
5. Y.A. Eracar, RAACR: *A Reconfigurable Architecture for Adapting to Changes in the Requirements*, master's thesis, Northeastern Univ., Boston, Mass., 1996.
6. K.J. Åström, *Adaptive Control*, Addison-Wesley, Reading, Mass., 1989.
7. C.E. Rohrs et al., "Robustness of Continuous-Time Adaptive Control Algorithms in the Presence of Unmodelled Dynamics," *IEEE Trans. Automatic Control*, Vol. 30, 1985, pp. 881–889.
8. J.S. Shamma, "Linearization and Gain-Scheduling," *The Control Handbook*, CRC Press, Boca Raton, Fla., 1996.
9. M.S. Branicky, B.S. Borkar, and S. Mitter, "A Unified Framework for Hybrid Control: Model and Optimal Control Theory," *IEEE Trans. Automatic Control*, Vol. 43, No. 1, Jan. 1998, pp. 31–45.
10. S. Chatterjee et al., *Modeling Applications for Adaptive QoS-Based Resource Management*, tech. report, SRI Int'l, Menlo Park, Calif., 1997.
11. R.J. Costello and D.-B. Liu, "Metrics for Requirements Engineering," *J. Systems and Software*, Vol. 29, No. 1, Apr. 1995, pp. 39–63.
12. M. Moriconi, X. Qian, and R.A. Riemenschneider, "Correct Architecture Refinement," *IEEE Trans. Software Eng.*, Vol. 21, No. 4, Apr. 1995, pp. 356–372.
13. D.C. Luckham and J. Vera, "An Event-Based Architecture Definition Language," *IEEE Trans. Software Eng.*, Vol. 21, No. 9, Sept. 1995, pp. 717–734.
14. P. Neumann, "System Development Woes," *Comm. ACM*, Vol. 40, No. 12, Dec. 1997, p. 160.

Mieczyslaw M. Kokar is an associate professor of electrical and computer engineering at Northeastern University. His research interests include formal methods in software engineering, intelligent control, and information fusion. He has an MS and a PhD in computer systems engineering from the Technical University of Wroclaw, Poland. He is a member of the IEEE and ACM. Contact him at the Dept. of ECE, Northeastern Univ., 360 Huntington Ave., Boston, MA 02115; kokar@coe.neu.edu; www.coe.neu.edu/~kokar.

Kenneth Baclawski is an associate professor of computer science at Northeastern University. His research interests include formal methods in software engineering, information retrieval, and database management systems. He has a BS from the University of Wisconsin and a PhD from Harvard University. He is a member of the IEEE and ACM. Contact him at the College of Computer Science, Northeastern Univ., 360 Huntington Ave., Boston, MA 02115; kenb@ccs.neu.edu; www.ccs.neu.edu/home/kenb.

Yonet A. Eracar is a PhD candidate in the College of Engineering at Northeastern University and is a software engineer at Teradyne Inc. His interests include software architectures, object-oriented design and modeling, compiler design, and code generators. He has a BS in electrical engineering and in physics from Bogazici University, Istanbul, and an MS in computer systems engineering and in engineering management from Northeastern University. Contact him at the Dept. of MIME, Northeastern Univ., 360 Huntington Ave., Boston, MA 02115; yeracar@coe.neu.edu; www.coe.neu.edu/~yeracar.

How to Reach Us

Writers
For detailed information on submitting articles, write for our Editorial Guidelines (m.davis@computer.org), or access http://computer.org/intelligent/edguide.htm.

Letters to the Editor
Send letters to

Managing Editor
IEEE Intelligent Systems
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
dprice@computer.org

Please provide an e-mail address or daytime phone number with your letter.

On the Web
Access http://computer.org/intelligent/ for information about *IEEE Intelligent Systems*.

Subscription Change of Address
Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *Intelligent Systems*.

Membership Change of Address
Send change-of-address requests for the membership directory to directory.updates@computer.org.

Missing or Damaged Copies
If you are missing an issue or you received a damaged copy, contact membership@computer.org.

Reprints of Articles
For price information or to order reprints, send e-mail to m.davis@computer.org or fax (714) 821-4010.

Reprint Permission
To obtain permission to reprint an article, contact William Hagen, IEEE Copyrights and Trademarks Manager, at whagen@ieee.org.

Intelligent SYSTEMS & their applications