Stochastic Optimal Switching: Theory, Architecture, Prototype

Yuanfang Cai, Kumar Manvendra, Pavel Sorokin, Kevin Sullivan

University of Virginia
Department of Computer Science
151 Engineer's Way
P. O. Box 400740
Charlottesville, VA 22903 USA
+1 804 982 2206
{yc7a, km4aj, ps7k, sullivan}@virginia.edu

ABSTRACT

Information hiding modularity (IHM) creates valuable options to switch one hidden module to another. The binding time for such switching is generally design time: as conditions change, new modules are created and switched in to keep a system current. Run-time switching is a clear analog and is sometimes beneficial. However, in face of switching costs and uncertainty about future conditions, when to switch is not clear. In recent work, we showed that it might be possible to estimate the economic value of design-time switching options created by IHM using real options theory. In this paper, we explore the flexibility to switch dynamically with a focus on optimal timing of decisions to switch. Our contribution is the idea that options theory holds an answer: Deciding when to switch under uncertainty is tantamount to deciding when to exercise a real option, which, in turn, is a special case stopping problem in stochastic optimal control. We give mathematical formulation of the problem and a reflective architecture for implementing our approach. As a feasibility test, we present a prototype that automates switching between dense and sparse matrix representations in an *open implementation* style based on estimates of the uncertainty in client behavior derived from runtime profiling. Our insight has potential relevance to the design of adaptive middleware and security and survivability control systems. The thought is to make internal switching decisions contingent on the ongoing resolution of monitored endogenous and exogenous uncertainties, such as the likelihood of system-level failure or security threat level.

Keywords

Information Hiding, Open Implementation, Real Options

1. INTRODUCTION

Recent works in software economics view software design as being to a significant degree a process of decision making under uncertainty and incomplete knowledge [12]. Also, it has been recognized that many important software design guidelines, such as information hiding and software architecture, risk-reducing spiral processes models and other phased project structures, as well as the guideline on the time of commitments to product and process design decisions, have useful interpretations in the realm of options theory [4][15], but are still covered more by folk wisdom and engineering rules of thumb than by science.

The problem for which we lack, and in this paper seek, a mathematical model is that of deriving optimal policies for on-line switching of the implementations in an *open implementation* context. To introduce this idea, we must first say a few words about options, and the interpretation of information hiding modularity in the context of options theory.

The idea of an option is simple: An option gives its owner the right without a corresponding obligation to make an investment (e.g., to buy or sell an asset) in the future on terms that are known today. Because the option holder need not exercise the rights in the option if the investment never becomes profitable, the worst-case payoff on an option is zero (and not negative, as can be the case for futures contracts, for example). However, if the right to invest might conceivably be profitable in the future, then, like a lottery ticket, the option will have some positive value today.

Figuring out how much an option is worth is the *option pricing* problem. In special cases, where the uncertain payoffs on an option is related to the uncertain payoffs on assets such as stocks that are already market-priced, an imputed market price for the option can be computed. If such an option is traded, and if its actual price diverges from its imputed price, unsustainable arbitrage opportunities will tend to drive the prices back together.

In other cases, an option has no market-enforced prices because the uncertain future payoff on the option is independent of the payoffs on market-traded assets. In this case, a rational option price has to be estimated by other means, such as decision-tree analysis with payoffs and probabilities that are subjective or derived from empirical data.

In recent work [13], we showed that information hiding creates options, and that pricing such options using a mathematical model introduced by Baldwin and Clark [2] leads to outcomes that are consistent with the findings in Parnas's seminal paper

[10]. The key idea in that work, and also in this one, is that information hiding creates options to replace one module implementation with another—options that can add significant value to system designs.

In this paper, our concern is not mainly with the options *pricing* problem, but with a related problem. Once a decision-maker holds an option, the question is when, if ever, is the best time to exercise the rights in the option to invest in the underlying asset? This is the *options timing* problem. The key insight from options theory in this regard is that it is not generally optimal to exercise an option as soon as the payoff is positive. Sometimes it is better to hold the option and wait to see how things turn out before deciding. This is the perspective that is most useful in this paper.

Returning to the traditional concept of information hiding, we note that the "binding time" for exercising the switching options created by information hiding is generally thought of as design time. As requirements and other parameters change, new modules are created and switched in to keep a system current. However, it is clear that information hiding also creates options to switch at later binding times, and that, in some cases, static design time switching is not sufficient to meet dynamically changing needs.

Kiczales et al., in work on *open implementation* [5], explored late-binding of decisions about which implementation to use. They saw that Black-box abstractions that bind implementation strategies early can create performance dilemmas for some clients: a given implementation behind an abstract interface may not be suited to all clients' needs. They thus proposed to give clients limited control over implementation strategy, through meta-level interfaces, so that clients could tailor underlying performance characteristics to their own needs without costly work-arounds.

With respect to the current work, the *open implementation* (OI) ideas have several shortcomings. First, although switching decisions are implemented at runtime, through meta-level interfaces, the bindings of these decisions are still generally made at design time: A client of a module is hardwired to call the module to select the client-appropriate implementation strategy. Second, the OI approach assumes that clients know the relevant usage patterns statically. However, the optimal switching policy will sometimes have to be dynamic, because of design-time uncertainty or ignorance about the relevant patterns. Third, even if a client wants to use OI meta-methods to switch implementation strategies dynamically, we lack a theory to guide the client to use an optimal switching policy. Finally, allowing clients to choose implementation strategies does compromise encapsulation, possibly complicating design evolution, and possibly causing other problems: e.g., local optimization can cause global degradation; space optimization can cause time deterioration; etc.

In this paper, we introduce the idea of, and an architectural design for, on-line stochastic optimal switching. This technique tackles the issue of the best time to switch in the face of non-trivial switching costs and uncertainty about future operating conditions. We make switching decisions based on historical usage patterns from which we infer expected future usage patterns. Also, we take switching cost into consideration. If a usage pattern is transient or if it fluctuates, it may not be worth it to pay the switching cost to seize a possibly ephemeral performance benefit.

We have shown that information hiding creates flexibilities which are tantamount to options. Similarly, the essence of the above question can be mapped to that of real options theory: having created options to switch using technologies such as *open implementation*, what is the best time to exercise runtime options? This mapping is feasible since the underlying nature of uncertainty, cost and benefit reflects the kind of problems that real options theory tries to solve.

Although it is too early to make strong claims, it appears that the insights in this paper have a potential to improve performance in such areas as adaptive middleware and security and survivability control [14]. The idea is that we might make internal switching decisions contingent on the ongoing resolution of monitored endogenous and exogenous uncertainties, such as the likelihood of system-level failure or the assumed security attack threat level.

In this paper, we use a very simple matrix example taken from Kiczales work on *open implementation* to test the feasibility of and to illustrate our ideas. We also give the mathematical real options model to illustrate the idea of using real options theory to determine best switch time in order to balance time/space tradeoff and switch cost. We realized this matrix prototype using AspectJ [8].

Section 2 introduces the matrix example, the *open implementation* solution and problems left unsolved by it, which motivates our dynamic switch model. Section 3 introduces the mathematical real options model. Section 4 gives the general architecture and the demonstration prototype. Section 5 evaluates the work and discusses the implications of this model to some related aspects. Section 6 concludes this paper.

2. A Matrix Example and its Open Implementation Solution

Let's assume that we are going to build an interactive Matrix product. Clients can add/remove elements from a matrix object. System performance is essential to user satisfaction. Two aspects of the performance are considered important and need to be balanced: element look-up time and memory usage efficiency.

Two most common implementations of matrices are 2D arrays, for dense, and linked list, for sparse matrices. The former, for dense matrices, has constant lookup time, but can waste memory space when most entries are zero. The latter strategy conserves memory spaces when matrices are sparse but generally incurs longer look-up times as the number of elements increases.

2.1 When to switch implementations?

In a traditional "black-box" design style, with no way to anticipate how customers will use the product, we would have to select one of the above methods and deliver the product. Recognizing the possible need for future changes, we might hide the matrix implementation as a secret behind an abstract Matrix interface, and take the two implementation methods as options. Changes might be triggered by customer's complaints about the time/space performance and implementation switch usually

happens in the next version. This approach would reflect the traditional black-box design style that Kiczales et al. criticized. Implementation strategy switching decisions are effected and bound at design time.

Unfortunately, changes in usage patterns could be both dynamic and hard to anticipate. Different clients may report different performance status. Even more dynamically, the same client might report different performance status during different periods of program execution. As a result, it might be hard or impossible to decide effectively when the "next version" should be launched and whether it is worth the cost to switch.

2.2 Open Implementation Solution

Having seen the problem caused by implementation preemption, Gregor Kiczales et al. proposed *open implementation* [7]. This method lets a client select an implementation method from a pool of implementation options, or even insert its own implementation in the lower level of the system in order to optimize its own performance. OI can even give clients the priority to change the semantics of a given interface. Taking the matrix product as an example, the client can call a matrix meta-method to choose or set an implementation according to its anticipated usage patterns.

However, the current OI method remains largely at design time: a client uses the meta-interface of an object to choose an implementation strategy before using the object. Changing implementations would often demand that code be changed and recompiled. Moreover, even though OI does, in principle, provide mechanisms to switch dynamically, in the form of strategy-selecting meta-methods, it provides no specific guidance on when, in general, it is optimal to switch. In addition, clients still have to anticipate the usage pattern, dense or sparse, of the matrix before they make any implementation decisions. Since our hypothetical example is an interactive matrix application, as we discussed above, the number of elements in it is determined by user inputs, and there is no way to anticipate in advance.

Other concerns include overall performance, integration and safety. Open implementation lets users optimize performance to fit their own needs, which may be at odds to overall performance. In addition, if an object is implemented in an ad hoc way, it would cause problems when some other objects want to integrate or communicate with it. Its partner may have to know its concrete implementation, which should be a "secret". Dynamic switching under principled program control could avoid these problems.

Although it is not difficult to change implementations dynamically at run time, a key problem remains: when to switch? Switching implementation strategies definitely will incur costs. In the matrix case, for example, the state of the current concrete representation of the abstraction would have to be extracted and used to initialize the newly switched-in representation.

Offline computation of fixed switching thresholds is an obvious and tried solution, but thrashing can occur if program behavior straddles the threshold. Imposing two thresholds to create a hysteresis band is a typical solution to this problem. One problem with this solution is that it does not take account of information that becomes available only at runtime. If program behavior is found to change smoothly and slowly, a narrow hysteresis band is appropriate. If behavior changes in a volatile way, a broader band is best. More generally, we lack a method for establishing switching policies that account explicitly and in principled ways for dynamic behavior. How, based on uncertain future conditions, can we select an optimal switching policy?

2.3 Dynamic Switching Based on Real Options Evaluation

In order to explore the possibility of optimizing overall performance in a more flexible, effective, and safe way, we propose Dynamic Option Switch Model in this paper. The main contributions of this paper are first, to show the feasibility of using real options theory as a basis for stochastic optimal dynamic switching, and, second, to exhibit a reflective system architecture for implementing such a technique.

Taking the real options terminology, we consider the different implementation methods, link list and array, as two options. The usage patterns depend on user input and present uncertainties and risks that will influence the value of the matrix product. The payoff of each method is represented by look-up time and memory usage percentage additively. Also we take the switch cost into account and measure that cost in terms of time and space.

The switch decision is based on history usage pattern, future usage probability, and the net performance gain computed from the underlying real options evaluation model. The essence of this computation is as follows: *Switch if and when the value of the immediate payoff of exercising the option exceeds the value of the expected payoff of waiting*. This policy is known from the theory of real and financial options [4] to be the optimal policy, in general, for deciding when to exercise an option on a risky asset; and it has been shown to have significant interpretations in several dimensions of software design and engineering discourse [12]. As a result, several key elements in our mathematical model are the history usage pattern, the future usage pattern prediction based on historical data, and cost of switching. The next section shows the rationale of this mathematical model in detail.

3. Dynamic Switch Based on Real Options Evaluation

We assume that the value of the matrix depends on both the element look-up time and the memory space usage efficiency. Look-up time affect the user response time. Similarly, memory usage efficiency also has impacts on system performance. If large arrays are declared but only a small portion of it is actually in use, as the number of rows/columns increase, the waste could be intolerable and decrease the value of the product. The single source of uncertainty that could affect the value of the product and that is relevant to switching decisions is the proportion of number of elements to the total number of cells in the matrix object.

To illustrate the problem in detail, we let l and m be the element look-up time and memory usage percentage respectively. Also we let $p(l)$ and $p(m)$ be the payoffs associated with look-up time and memory usage percentage (that is, the proportion of memory actually used for elements to the total memory claimed for the whole matrix. For simplicity, we don't take the totally used memory as a measurement, that is, we ignore such overhead as pointer fields that the link list might have). We denote the value

of the product using rigidly one of the implementation methods, link list (S, for sparse) or 2D arrays(D, for dense), as V(S) and V(D), respectively. Let *i* stand for the implementation of the matrix: either *S* or *D*. As a simple model of the value of the system as a function of its constituent properties, we will assume that the value of *V(i)* is the sum of the payoffs, $p_i(l)$ and $p_i(m)$, on using implementation *i*: $V(i) = p_i(l) + p_i(m)$.

Initially, if we implement the matrix using a linked list, the payoff due to memory usage percentage, $p_S(m)$ can be back-of-the-envelope approximated as a constant value, since the matrix will use up all the memory it declared. As the proportion of number of elements to the total number of cells increase, the look-up time will go up, and make $p_S(l)$ go down. As the result, V(S) will go down accordingly. If V(S) goes down to an intolerable level, we might consider change the implementation to 2D arrays. This scenario is shown in figure 1:

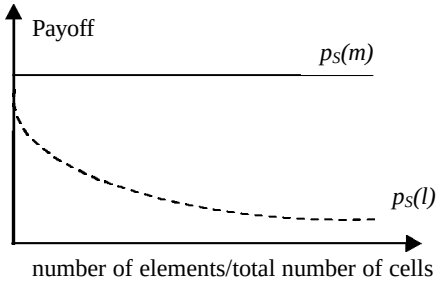

Figure 1: The payoffs of the matrix implemented by link list

Instead, if we implement the matrix using 2D arrays, the $p_D(l)$is a constant value, since the look-up time for any elements would be the same. As the proportion of number of elements to the total number of cells decrease, the wasted memory will go up, and make $p_D(m)$ go down. As the result, V(D) will go down accordingly. If V(D) goes down to an intolerable level, we might consider change the implementation to link list. This scenario is shown in figure 2:

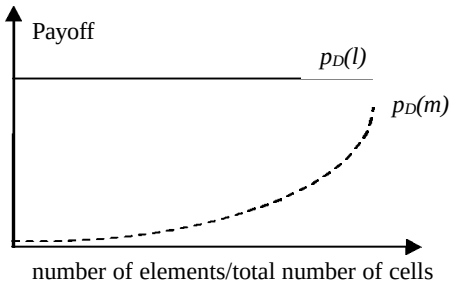

Figure 2: The payoffs of the matrix implemented by 2D arrays

The volatility, that is, the proportion of number of elements to the total number of cells, can be modeled as a continuous stochastic process denoted by σ. In our example, we have modeled the volatility dynamics as a piecewise constant process within a time interval, but from one interval to another, the volatility is stochastic [11]. Due to this piecewise constant approximation, the volatility evolves as a geometric Brownian motion and can be approximated as a first order autoregressive process as follows:

Let x(t) be the user response at time t, and the return at time t with respect to a time interval ΔT can be represented as[3]: $g_{\Delta T}(t) = x(t) - x(t - \Delta T)$, and

$$s(t) = \frac{1}{N}\sum_{k=1}^{N} | g[\Delta T](t-(k-1)\Delta T) |$$

T is the sampling period, and N is a positive integer such that $T = N\Delta T$. The parameters T, N, and t may vary over large ranges.

At any time, that uncertainty can be represented by two alternative states: getting denser or sparser. If it goes denser, V(S) will decrease by d, while V(D) will increase by d. Similarly, if it goes sparse, V(S) will increate by u, while V(D) will decrease by u. We use "+" and "-" to denote the states in which the value goes up or down by *u* and *d* respectively.

However, we must restrict the values and probabilities of the two states in such way that the expected value return over the next time interval, Δt, is equal to rΔt and that its volatility is $\sigma\sqrt{\Delta t}$ [11]. Here r is the risk-free interest rate. As a result, we get the following equations:

$$\frac{puV(i)+(1-p)dV(i)}{V(i)} = e^{r\Delta t} \quad (1)$$

$$pu^2 + (1-p)d^2 - [pu + (1-p)d]^2 = s^2\Delta t \quad (2)$$

The probability, *p*, weights the likelihood that the matrix will become denser or sparser to obtain the risk-free rate of returns and is called the risk-neutral probability [1]. We can measure the volatility, σ, using history data as we discussed above and compute u, d and p accordingly:

$$u = e^{s\sqrt{\Delta t}}$$

$$d = e^{-s\sqrt{\Delta t}}$$

$$p = \frac{e^{r\Delta t} - d}{u - d}$$

We use a binomial lattice to illustrate the value of the matrix at the end of each interval. For simplicity, we assume two periods of time. Let $V_t^s(i)$ represents the value of the matrix at time t, state s and using implementation i, (where i = S or D and s = + or -)

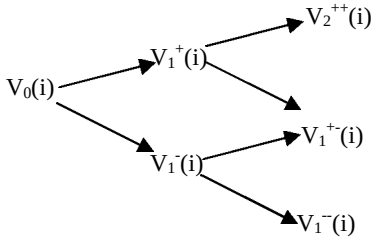

Figure 3: Each node represents the value of matrix implemented by method i at each end of time period and at each state.

At the end of any time period t, state s, suppose the matrix is implemented using link list, we consider the value of switching from S to D, which is denoted by $S_t(S \rightarrow D)$.

If these is no switching cost, $S_t(S \rightarrow D) = \max(V_t^s(D) - V_t^s(S), 0)$. At any time, if we observe that $S_t(S \rightarrow D)$ is larger than 0, we change implementation to 2D arrays. In addition, we can easily show that this switch has no effect on subsequent payoffs [11].

However, switching has costs that can not be ignored. When the link list is switched to a 2D array, time and memory costs occur. The time cost is the time to look up all elements in the list and copy them to the array. The memory cost can be measured by the memory claimed but unused. We denote the total cost as *C(S→D).*

When the 2D array implementation is switched to link list, only time cost occurs, which includes the time used to copy elements into link list and the accompanying link list look-up time. We denote the total cost as *C(D→S)*

Furthermore, taking the switching with costs not only affects the current decision and payoff but also alter the switching decisions in future periods since it would affect the mode under which the system would operate as it enters future periods and hence affect the future switching cost [11]. In this case, we have to compute expected future benefits, necessitating the use of backward dynamic programming process.

As of time t, given that state s is entered while operating in mode i (S or D), let $F_t^s(i)$ denote the value of the matrix with dynamic switching ability, and let $V_t^s(i)$ denote the value when operating in mode i. E[.] is the risk-neutral expectation operator.

At any time, we have an option: continue using the current implementation for one more period, with value $V_t^s(i)$ plus expected future benefits, or switch to the other implementation, paying the specified switching cost and getting the value of the other implementation and its expected future benefits.

The expected future value can be computed by:

$$E[F_{t+1}^s(i)] = p\, F_{t+1}^+(i) + (1 - p)\, F_{t+1}^-(i)$$

A mode switch will be optimal only if the value from switching immediately exceeds the value from delaying potential switch—a value that reflects the opportunity cost of switching too soon.

For example, if at time t, state s and mode S, the value from delaying is represented by:

$$V_{delay} = V_t^s(S) + E[F_{t+1}^s(S)]/(1+r)$$

The value from switching immediately is represented by:

$$V_{exe} = V_t^s(D) + E[F_{t+1}^s(D)]/(1+r) - C\ (S \rightarrow D)$$

the value of the flexible matrix at time t is:

$$F_t^s(S) = \max(V_{delay}, V_{exe})$$

$$= \max\{V_t^s(S) + E[F_{t+1}^s(S)]/(1+r),$$

$$V_t^s(D) + E[F_{t+1}^s(D)]/(1+r) - C\ (S \rightarrow D)\} \quad (3)$$

So, the optimal time to exercise the option is when V_{exe} exceeds V_{delay},. The backward iterative process would start from the assumed terminal time T. This terminal time could be the expected life time of the product. Since there is no future expected benefit,

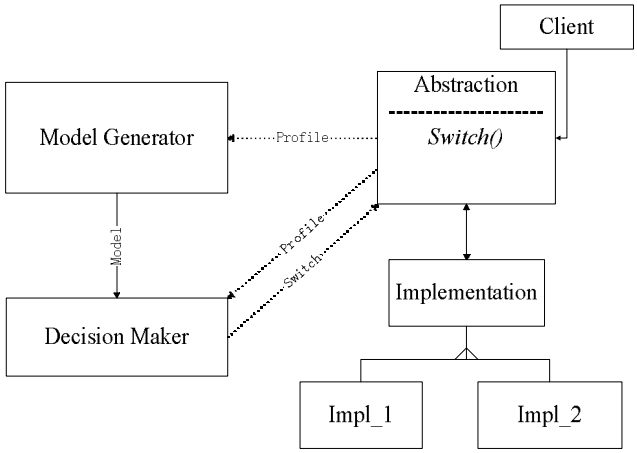

Figure 4: The reflective switching architecture

$$V_{delay} = V_T^s(S)$$

$$V_{exe} = V_T^s(D) - C\ (S \rightarrow D), \text{ and}$$

$$F_T^s(S) = \max\{V_T^s(S), V_T^s(D) - C\ (S \rightarrow D)\}$$

From now on, we can compute backwards from $F_{T-1}^s(S)$, $F_{T-2}^s(S)$… to $F_t(S)$, at current time t, and make the appropriate decision – to switch or not to switch.

4. Architecture and Demonstration Prototype

We now present a reflective architecture for implementing the real options approach to stochastic optimal on-line switching. We also describe a realization of the architecture for the matrix example. Figure 4 illustrates the components and their relationships. The architecture has five component types:

- *Client:* the source of the uncertain future conditions on which the switching decision is to be based;
- *Abstraction*: an abstract component supporting several dynamically interchangeable concrete *Implementations;*
- *Implementations*: the set of interchangeable realizations of the *Abstraction;*
- *Model Generator*: observes the evolving load imposed by the *Client* on the *Abstraction* and produces a model of that load as a stochastic process, focusing on its variance over time, in particular;
- *Decision Maker*: receives the stochastic model of the *Client* from *Model Generator*, observes the *Client*, and, in light of both, makes and effects switching decisions.

The relationships in this abstract system are as follows:

- *Client* places an uncertain evolving load on *Abstraction.*
- *Model Generator* observes the load and models it as a stochastic process.
- *Decision Maker* uses both the generated model and observations of the *Client* load to decide when, if ever, to switch. It effects a switching decision by calling a *Client* meta-method, in an *open implementation* style.

We have tested the feasibility of the architecture by using it to managing on-line switching for the matrix case. We implemented the system in AspectJ. Aspect provides background threads that don't disturb the main computation until it decides to switch, and reflective computation, which helps us collect ongoing usage data.

The matrix *Implementation* uses an abstract factory pattern to generate dense and sparse concrete matrix representations. We implemented the matrix *Abstraction* as a class having a matrix *Implementation* as a state component and a method for switching representations. We implemented the matrix *Model Generator* as a per-object aspect, activated by calls from matrix *Client* objects to matrix objects. The matrix *Model Generator* tracks the density of non-zero matrix entries and models this time-varying load as a stochastic process. When elements are added or removed, data is collected and, periodically, the parameters p, u and d are recomputed based on the observed volatility in the load, σ. The matrix *Decision Maker* periodically takes p, u and d as parameters and observations of the actual load on the matrix *Abstraction*. Based on this information, it computes future expectation payoffs, $E[V_t^s(i)]$. Using this value, it then computes the immediate payoff, V_{exe}, and the value of delay, V_{delay}, and makes a decision to switch or not.

5. Evaluation

The contribution of this paper is the idea that options theory can provide a sound basis for optimal timing of runtime decisions to exercise switching options in the context of *open implementation* architectures, using models of the uncertainty of future operating conditions derived from observations of past operating conditions. We also presented a simple reflective architecture that shows how modern reflective programming constructs can be used both to collect the data needed for model create and to control underlying implementations strategies.

Our argument for the *validity* of this perspective rests on the clear mapping of information hiding modularity to the creation of switching options, an established idea [5][13]; on the established theory of optimal timing of decisions to exercise options [4][15]; and on the established mapping of that theory to a variety of problem formulations in software engineering, including information hiding modularity and various problems of optimal timing of design decisions in development processes [12].

The key insight in this paper is that we can employ the existing work on timing to the problem of *on-line switching*. The value of this insight lies in the combination of the generality of the basic formulation and the anticipated utility of dynamic switching as a mechanism. In terms of generality, an options formulation applies to any (runtime) switching decision for which switching costs are non-trivial and future operating conditions and thus payoffs are uncertain. The anticipated utility of dynamic switching is not a topic that we explore in any depth or generality in this paper. It is likely, however, to be at the heart of many dynamically adaptive systems, including adaptive quality-of-service architectures, adaptive middleware layers and survivability control systems [14].

Today, switching policies for such systems are usually computed off-line based on static assumptions about load factors. It not clear that such policies are optimal, especially when actual load characteristics diverge from the anticipated. This work provides a basis for on-line derivation of models from which optimal policies are derived, and the derivation is based on the sound mathematics of options timing, a special case *stopping problem* in stochastic optimal control.

On the other hand, we recognize our work as early exploration of a new area, with much remaining to be done.

First, the conditions under which the load factors that drive switching decisions have reasonable stochastic models is unclear. When we use equation (1) and (2) to compute u, d and p, we tacitly assumed that uncertainty factor, the number of elements/the number of total cells, has lognormal distribution—that extreme cases are rare—and that the load imposed by the client moves like a stock price (in a geometric Brownian motion).

Whether this is a reasonable assumption depends on the behavior of the client. We specified a demonstration client to have this property, but the extent to which real loads exhibit such regularity, and the kinds of regularities they exhibit, remain as issues to be explored. Some load factors probably do behave like Brownian motions and others like Poisson processes, for example. In some cases, a process behaves as white noise, in which case no useful regularity is present.

The key idea in this paper is that if there is any statistical regularity to the underlying stochastic process, then the problem of deciding when to switch *is* one of options timing, independent of the specific kind of underlying distribution or of its parameters. The main idea is widely applicable: whenever we are faced with switching, cannot ignore costs, can analyze the underlying processes through data profiling, can value expected future payoffs, and can take switching costs into account, we need to compare the value of waiting (the opportunity cost of switching) to the payoff on switching to decide whether switching is optimal.

Remaining challenges in this area include characterizing important stochastic behaviors in real systems empirically, and deriving models for such distributions. In general, the derivation of stochastic models from sample data is a problem in time series analysis, and is studied extensively by econometricians, among others. We plan to begin such studies use profiling techniques to analyze distributions from real systems. One case study involves adaptive real-time operating systems, being explored at the University of Virginia, in which scheduling policies can be changed dynamically according to current CPU usage.

Second, reflective computation, especially for profiling purposes, is not free, and can be expensive or even infeasible, e.g., in some distributed systems. The costs of runtime monitoring and model derivation include both design- and run-time costs. On-line decision-making is just more complex and incurs more overhead than back-of-the-envelope off-line computation of switching thresholds. We clearly need to do experimental systems work to explore the tradeoffs in this dimension.

Third, mapping computational costs and benefits to a scalar-valued abstract numeraire (an analog of money) is necessary if an options formulation is to be employed; but we don't know how to do it well, in general. When implementing our prototype, we determined the payoffs associated with look-up time and memory usage percentage, *p(l)*, and *p(m)*, in a not wholly unreasonable,

but still rather arbitrary, way. It's also not entirely clear how to set the risk-free discount rate in the options formula, which models the time value of money (the interest one could earn by lending), in a reasonable way. There has been significant work on economic markets for resource allocation in distributed systems, exemplified that of Miller and Drexler [9]. Financial models are applied in this work to make technical decisions. We plan to explore how to adapt the techniques developed in such work to our application.

Finally, we recognize that our matrix application is somewhat artificial. We chose it because it has figured prominently in *open implementation* design discussions. We understand that many matrix clients exhibit either dense or sparse behaviors, never transitioning from one to another. The example is but a proxy for systems in which a stochastic process drives switching decisions.

6. Conclusions

Information hiding creates options. Today, we lack models to guide us in making the best decisions about when to exercise such options in the face of switching costs and uncertainty about relevant future conditions. In this paper, we explored the application of options theory and decision making when *dynamic* switching is needed for such purposes as on-line performance optimization and adaptive control.

The essence of the idea is that; when we are at such a decision point, we should first analyze the underlying uncertainty distribution and volatility according to the data collected in the past. Based on that, we can compute the expected future benefits both when remaining unchanged and switching to another mode. If the value of delay exceeds the value of switching minus the costs, stay on the current mode; otherwise, change to another one. We know from options theory, as a branch of stochastic optimal control, that this is the right way to think about the issue: exercising an option incurs an opportunity cost, related to the characteristics of the underlying stochastic process. The payoff of exercising immediately should not only be positive, but balance that opportunity cost.

We also presented a reflective architecture in terms of which we can begin to explore this insight experimentally. We showed that a demonstration prototype can be implemented in this style using advanced reflective constructs of aspect-oriented languages. In particular, our matrix prototype, programmed in AspectJ, switches implementations based on backward dynamic programming to compute system values at each time period and make decisions.

In conclusion, we note that the approach that we have presented attacks a vexing problem facing options-theoretic approaches to software design and engineering: where to get the data to calibrate options models. The key idea in this paper is that whether or not to switch is often contingent on conditions that are subject to on-line profiling. Examples include unpredictable but stochastically regular loads imposed by clients on servers, as explored in this paper; and, perhaps, exogenous conditions such as threat levels to which a system should adapt by trading off functionality and responsiveness for security and survivability. We plan to explore this issue by integrating the work reported here with earlier work on survivability control systems [14], which was based on on-line dynamic switching, but which, to now, has been without a theory for how best to do that.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grant ITR-0086003. Discussions with graduate students in the University of Virginia Software Economics and AOSD Seminars, CS 851 and CS651, in the Spring semester of 2002, were inspiring and helpful. Alfredo Garcia pointed us to the econometrics literature for deriving models from time series data.

NOTE TO REVIEWERS

The source code for our demonstration system will be available for review on request, and will be made public pending obligatory intellectual property disclosures at the University of Virginia. Please contact Kevin Sullivan by e-mail through Mary Lou Soffa, the FSE PC co-chair, if you wish the review the design and code.

REFERENCES

[1] Amram, M. and Kulatilaka, N., *Real Options: Managing Strategic Investment in an Uncertain World*, Harvard Business School Press, 1999

[2] Baldwin, C. Y. and Clark, K. B., Design Rules: The Power of Modularity, MIT Press, 2000.

[3] Breymann, W, *Prediction of Market Volatility with a Cascade Model*, http://www.physik.uni-bielefeld.de/complexity/Cascade_Model_Breymann.pdf, talk in Maths Week, London, 26 - 30 Nov 2001

[4] Dixit, A., and R. Pindyck, 1994, "Investment under uncertainty", Princeton Univesity Press.

[5] Favaro, J.M., K.R. Favaro and P.F. Favaro, "Value Based Software Reuse Investment," *Annals of Software Engineering 5*, 1998, pp. 5 – 52.

[6] Kiczales, G., Ashley, J.M., Rodrigues, L., Vahdat, L. and Bobrow, D.G., "Metaobject protocols: Why we want them and what else they can do", *Object-Oriented Programming: the CLOS Perspective*, pages 101---118, MIT Press, Cambridge, MA, 1993

[7] Kiczales, G., "Beyond the black box: Open implementation," *IEEE Software*, January 1996.

[8] Kiczales,G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W.G. Griswold, "An overview of AspectJ," in *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP), 2001.

[9] Miller, M. S. and Drexler, K. E., "Markets and Computation: Agoric Open Systems" in *The Ecology of Computation*, Bernardo Huberman (ed.) Elsevier Science Publishers/North-Holland, 1988.

[10] Parnas, D. L., "On the criteria to be used in decomposing system into modules," Communications of the ACM, Vol. 15, No. 12, December 1972 pp. 1053–1058.

[11] Schwartz, E.S. and Trigeorgis, *Real Options and Investment Under Uncertainty*, MIT Press, 2001

[12] Sullivan, K.J., P. Chalasani, S. Jha and V. Sazawal, " Software Design as a Investment Activity: A Real Options Perspective," in Real Options and Business Srategy: Applications to Decision Making, L. Trigeorgis, consulting editor, Risk Books, 1999. (Previously Sullivan et al., " Software Design Decisions as Real Options," Technical Report 97-14, University of Virginia Department of Computer Science, Charlottesville, Virginia, USA, 1997.)

[13] Sullivan, Griswold, Cai and Hallen," The structure and value of modularity in software design", ESEC/FSE 2001.

[14] K.J. Sullivan, J.C. Knight, X. Du and S. Geist, "Information Survivability Control Systems," *Proceedings of the 21st International Conference on Software Engineering*, pp. 184--193, May 1999.

[15] Trigeorgis, L., *Real Options: Managerial Flexibility and Strategy in Resource Allocation*, MIT Press, 1997