An Options Approach to Software Prototyping

Prasad Chalasani
Carnegie Mellon University
Pittsburgh, PA 15213-3890, USA
+1 412 268 3194
chal@cs.cmu.edu

Somesh Jha
Carnegie Mellon University
Pittsburgh, PA 15213-3890, USA
+1 412 268 3740
sjha@cs.cmu.edu

Kevin Sullivan
University of Virginia
Charlottesville, VA 22903, USA
+1 804 982 2206
sullivan@cs.virginia.edu

ABSTRACT
Prototyping is often used to predict, or reduce the uncertainty over, the future profitability of a software design choice. Boehm [3] pioneered the use of techniques from Bayesian decision theory to provide a basis for making prototyping decisions. However, this approach does not apply to situations where the software engineer has the flexibility of waiting for more information before making a prototyping decision. Also, this framework only assumes uncertainty over one time period, and assumes a design-choice must be made immediately after prototyping. We propose a more general multi-period approach that takes into account the flexibility of being able to postpone the prototyping and design decisions. In particular, we argue that this flexibility is analogous to the flexibility of exercise of certain financial instruments called options, and that the value of the flexibility is the value of the corresponding financial option. The field of real option theory in finance provides a rigorous framework to analyze the optimal exercise of such options, and this can be applied to the prototyping decision problem. Our approach integrates the timing of prototype decisions and design decisions within a single framework.

KEYWORDS
Software engineering economics, prototyping, real options, financial option theory, sequential investment.

1 Introduction

Although much progress has been made in the identification of key software design paradigms and concepts (such as information-hiding, prototyping, reuse, and program-families), the *criteria* for making decisions about how to use these concepts have largely been ad-hoc. For instance, consider a software engineer faced with the decision of whether to change the implementation of a cache-coherence protocol in a certain distributed database management system. The economic profitability of the new protocol depends on which of two possible future scenarios will occur. In one scenario, the new protocol would work extremely well, whereas in the other, it might degrade performance. The two scenarios are perhaps estimated to be equally likely, but we do not know which one will occur. Prototyping can be a way to predict (or reduce our uncertainty over) which scenario will occur. Based on the prototype the software engineer can perhaps revise his estimate of the probability of the favorable scenario to 70%, and then decide whether or not to change the protocol. Of course constructing and evaluating the prototype involves an expenditure of limited resources such as programmer time, CPU time and memory space. The software engineer must therefore weigh the cost of prototyping against the benefit of prototyping. The benefit of prototyping (as we show in an example later) is the increased expected payoff from a more informed decision based on prototyping, compared to a decision without prototyping.

Boehm [2, 3] was among the first to propose an appealing economics-based approach to providing a sound basis for software decisions in general and prototyping decisions in particular. His approach views software decisions as problems of investing in the face of uncertainty over the future payoffs from the investment. In his model, the effect of prototyping is to change the conditional probabilities of adopting each design alternative given that the future is one of a certain set of possible scenarios. He applies techniques from Bayesian decision theory to calculate the expected payoff from a prototyping decision.

Although the Bayesian approach recognizes the importance of uncertainty in modeling prototyping decisions, it has some limitations. To begin with, this approach is based on the traditional net present value (NPV) analysis found in business textbooks: First calculate the present value of the future benefits that the investment (i.e. prototyping and the consequent design choice) will generate. If the future benefits are uncertain, compute the expectation of this present value. Next calculate the (expectation of the) present value of the prototyping and design cost. The NPV is the first quantity minus the second, i.e., benefit minus cost. The software engineer then applies the NPV rule traditionally found in business textbooks (e.g., [4]): if the NPV is positive then proceed with the investment, otherwise don't. However, this simple rule is often suboptimal, since it is founded on a faulty assumption. It views the investor's (or software engineer's) decision as a *now-or-never* decision, in the sense that if the prototyping is not done now, the opportunity is lost for ever. Thus if the only possibilities available to the investor are to invest now or invest never, then indeed he or she should invest if and only if the NPV is positive. However, most investment decisions, including those in the software area, can be *delayed*. This is especially true when a certain implementation of the software is already in place, and a design *change* is being considered. This brings into play several other possible strategies that must be considered when picking the optimal one. Specifically, the investor's decision can be *contingent* on what kind of future unfolds. For instance the investor can wait for a month, and decide to not invest if the next month reveals information that indicates that there would be no net positive payoff from the investment. Or perhaps after a month there is less uncertainty about the profitability of the investment. This type of wait-and-see strategy can have a larger expected payoff (even when discounted to the present time) than investing right away, so that even when the above NPV rule indicates an investment, it would be suboptimal to invest right away. Of course, if the expenditures can somehow be recovered when conditions turn sour, then the decision to invest is effectively reversible, and in this case it would suffice to consider this as a now-or-never decision. However, the expense of undertaking a software design decision cannot usually be recovered. In this sense software decisions, including prototyping decisions, are *irreversible* but capable of being delayed.

Another limitation of the statistical decision theory framework for prototyping is that it only assumes one period of uncertainty, and assumes that both the prototyping and design decisions are made in this period. However in reality, once the prototyping is done, the software engineer can wait several time periods during which additional uncertainties might be resolved before making his design choice (especially if a design *change* is being considered).

In this paper we view the prototyping and design decisions not merely as investments under uncertainty (as previous work has done), but more specifically as *irreversible but delayable investments under uncertainty*. In a growing body of research [6, 7, 8, 9, 12], this viewpoint is being increasingly accepted as the appropriate one (as opposed to the traditional NPV rule) for analyzing capital investments by companies in the face of an uncertain future. Rather than viewing investments as being either now-or-never propositions or as being reversible, this approach stresses the fact that companies have *opportunities* to invest, and that they must decide how best to exploit these opportunities. At any time, the company has the *option* to make the investment: it has the opportunity, but is not obligated to, make the investment. The success of this new research is based on the important observation that investment opportunities are analogous to financial *call options* (see Section 2.2 for details), and that when a company makes an irreversible investment, it in effect "exercises" its call option. In fact, to make the analogy explicit, the investment opportunity available to the company has been called a "real option". A real option is the flexibility a manager has for making decisions about *real* assets (in contrast to shares of stock) [11]. Thus the company's investment problem is essentially one of optimally exercising its call option. The valuation and optimal exercise of financial options has been an active area of research in finance over the past two decades. Researchers have been applying results from the well-established area of options to capital investment problems, and this has lead to the new field of *real option theory.*

In this paper we are advocating a similar use of the options approach to study software prototyping and design decision problems. This has the benefit that we can draw upon the research done in the area of real options. A preliminary paper outlining this approach was presented by one of the authors at the 1996 Software Architectures Workshop [13]. A more detailed paper that applies our ideas to software design decisions appears in [14]. Withey [15] also applies real options ideas to analyze strategies for building reusable software modules. Baldwin and Clark [1] have used the theory of real options to quantify the benefits of modularity in terms of increased design flexibility. The present paper focusses on developing models for analyzing prototyping strategies as well as design strategies that are influenced by prototyping. Our basic premise is that prototyping, unlike other software design decisions, affects our beliefs about the probabilities of future events or "states of nature". The models just mentioned therefore cannot be used to model the effects of prototyping. An additional complication in the current paper is that we study the optimal timing of both the prototyping decision *and* the resulting design decision in the same framework. (Our earlier paper [14] alludes informally to the possibility of modeling multiple decisions.) Thus in the present paper we need a significantly richer model.

Our proposal is that formulating software prototyping criteria in terms of real options addresses the limitations of earlier work on prototyping. Moreover, this approach can lead to previously unrecognized design principles. Our goal is to show that real options theory can help describe, and perhaps even prescribe, software prototyping and design decision-making behaviors.

The rest of this paper is organized as follows. Sections 2.1 and 2.2 introduce the necessary concepts from probability theory and financial options. Section 3 describes the basic idea of real option theory and how to apply it to software design decisions. Section 4.2 present our model of prototyping and design-decisions based on the options viewpoint. Section 5 concludes with a discussion of the contributions of this paper, and future research directions.

2 Preliminaries

2.1 Probability concepts

In this section we introduce some standard mathematical terminology and notation that is used in the paper. Much of this material appears in our earlier paper [14]. We will model future uncertainty by means of a discrete **decision tree** of finite depth N, where N represents the maximum number of future time steps (e.g. months, years, etc) that we wish to model. The root node is considered to be at depth 0 and represents the present time, i.e., time 0. The nodes at depth k represent the possible states of the world at time k. The children of a depth k node v are the possible next-states at time $k+1$, given that the state at time k is v. If a node w is a descendant of v, we write $v \rightarrow w$. For $k = 0, 1, \ldots, N$, a **random variable** X is a mapping (or function) that associates with each node v, a real number $X(v)$. A random **process** is a sequence of random variables $\{X_k\}_{k=0}^{N}$, (often referred to briefly as the "process X_k") where for each k, $X_k(v)$ has non-zero values only for nodes at depth k. When we want to refer to the value of a random process X_k at a specific node v, we will often drop the subscript and just write $X(v)$. In any decision tree, we define the special random variable $\delta(v)$ to be the depth of v.

As an illustrative example, it is useful to have the following simple decision tree, called the **binomial tree**, in mind. Imagine we toss a coin N times. Each non-leaf node in this tree has two children. Each of the 2^N paths in the tree represents a particular sequence of coin-toss outcomes. On any path, for $k = 1, \ldots, N$, the k'th branch is an up-branch if the k'th coin-toss lands heads (H), and it is a down-branch if it lands tails (T).

With each branch in an decision tree, we associate a **probability** in such a way that the sum of the probabilities of the branches emanating from a given node is 1. For instance if we toss a fair coin in the example above, the probability of each branch is 0.5. For any node v in the decision tree, the probability that state v will occur, denoted $\mathbf{P}(v)$, is computed in the obvious way: multiply the probabilities of the branches from the root node to v. The **expectation** of a random variable X, denoted by $\mathbf{E}(X)$, is defined as

$$\mathbf{E}(X) = \sum_{v} X(v)\mathbf{P}(v). \tag{1}$$

The concept of conditional expectation is an important one for this paper. Let us imagine we are in a particular state of the world at time k, i.e., at a particular node v at depth k. Then the **conditional expectation** of a random variable X, *given* that we are at v, is denoted by $\mathbf{E}(X|v)$, and is defined as

$$\mathbf{E}(X|v) = \sum_{w: v \rightarrow w} X(w)\frac{\mathbf{P}(w)}{\mathbf{P}(v)}, \tag{2}$$

which is just a form of the familiar Baye's rule. Clearly this conditional expectation will in general be different from $\mathbf{E}X$, and will depend on which depth-k node v we're at. For instance in the coin-toss tree above, suppose the random variable H_k is the number of heads up to time k, on the path to a specific node in the tree. Then if v is at depth k and $m \geq k$, the conditional expectation $\mathbf{E}(H_m|v)$ will be higher if the path to v consists of more heads.

We will use decision trees in this paper to model the timing of various investment decisions. For instance we might want to decide *when* (i.e. at which nodes in the tree) and *how much* to invest in building a software prototype. In general let us assume that at any node we are allowed c possible levels of investment, numbered $1, 2, \ldots, c$. We consider level 0 to represent no investment. A **decision rule** τ with respect to a decision tree T is a mapping from the set of nodes of T to the set $\{0, 1, \ldots, c\}$, with the restriction that on any path of the tree, there is at most one node v with a non-zero value of $\tau(v)$. In other words, a decision rule specifies a rule that we can follow as the state of the world changes along the tree. Whenever we are in a state v for which $\tau(v) \neq 0$, we invest at level $\tau(v)$. In the coin-toss binomial tree, an example of a decision-rule is: "invest at level 1 when the coin has landed heads 3 times", or more formally: $\tau(v) = 1$ if $H(v) = 3$, and $\tau(v) = 0$ otherwise.

2.2 Financial Options

We now describe some basic concepts in option theory. For further details the reader is referred to Hull's [10] excellent introductory text.

The simplest kinds of options are call options. An **American call option** on a certain stock is a financial contract with the following features: it gives the holder of the contract the *right* but not the obligation to buy a share of the stock at a fixed price called the **strike** (or **exercise**) price L from the writer (i.e. seller) of the contract, on or before a certain **expiration** date of T time units. The holder thus has the "option" of deciding whether or not to exercise the contract, i.e., demand a share of stock at the strike price L from the contract writer. This is why the contract is called an option. When the option is exercised or the option expires, the option ceases to exist. Thus option exercise is *irreversible.*

In order to discount future cash flows to the present time, we will need to assume that money can be borrowed or lent (for example, via a bank or government bond) at a risk-free interest rate of r. Thus a dollar lent or borrowed at (discrete) time k is worth $R = 1 + r$ dollar at time $k + 1$. It is common to refer to R as a **discount factor** since a dollar at time k, discounted to the present time (i.e. time 0) is worth $1/R^k$.

To describe an American call option formally, we will model the price of the underlying stock in terms of a depth-N decision-tree. The time N corresponds to the expiration of the option. We let $\{S_k\}$ be the random process that denotes the price of the underlying stock. It is clear that the holder should not exercise the option at a node v if $S(v) \leq L$. On the other hand, if $S(v) > L$, the holder can (but is not obligated to) exercise the option, and if she does, the option writer is obligated to sell her a share of stock at the strike price L. The holder could then immediately sell the share in the market at $S(v)$, and make a profit of $S(v) - L$. Thus the profit that can be realized by exercising the American call option at time k is $\max(S(v) - L, 0)$, which we refer to as the **payoff** $G(v)$ from the option. It is standard notation to denote $\max\{x, 0\}$ by x^+, so we can write the payoff as the random variable

$$G_k = (S_k - L)^+, \tag{3}$$

or in other words, for any node v, the payoff $G(v) = (S(v) - L)^+$.

What is the best exercise strategy for the holder of an American call option, if she is still holding it at time k? Any exercise strategy can be described by a decision rule (defined in the previous section) τ that maps nodes to the set $\{0, 1\}$. If we are in state v, we exercise if and only if $\tau(v) = 1$. An example of an exercise strategy is: "exercise when the stock price exceeds a certain threshold λ, or the expiration date is reached", and is described by the decision rule τ where $\tau(v) = 1$ if $S(v) \geq \lambda$ or the depth $\delta(v)$ of the node v is N, and $\tau(v) = 0$ otherwise.

For a given exercise decision rule τ, and a given node v at depth k, the expected value of the strategy τ discounted to time k is denoted by V_k^τ, and is computed as follows. At any node w that is a descendant of v in the tree, if the option is exercised (i.e., $\tau(w) = 1$), the payoff is $G(w) = (S(w) - L)^+$, and if it is not exercised the payoff is 0. Thus in general at any node w we can write the payoff as $G(w)\tau(w)$, which is worth $G(w)\tau(w)R^{k-\delta(w)}$ at time k. Therefore the expected value of the strategy τ, discounted to time k, given that we are at a node v, is

$$V_k^\tau(v) = \mathbf{E}\left(G\tau R^{k-\delta}\middle| v\right). \tag{4}$$

Our option holder would of course want to choose the strategy τ so that this expectation is maximized. We denote this maximum by the random variable V_k:

$$V_k(v) = \max_\tau V_k^\tau(v). \tag{5}$$

In other words, V_k is the best expected present value at time k realizable over all possible exercise strategies. We refer to this value as the **option value** at time k, for reasons that will become clear shortly.

Since immediate exercise is a valid strategy at any time, the option value $V(v)$ must be at least as large as $(S(v) - L)^+$. In fact, if $(S(v) - L)^+ < V(v)$, this means that the immediate exercise strategy is *not* optimal, and that some other strategy will yield a strictly greater expected present value of payoff, under our assumed stock price model. Thus in this situation, it is beneficial to not exercise and wait. On the other hand, if $(S(v) - L)^+ = V(v)$, then there is nothing to be gained in waiting, at least under our assumed stock price model. In this case it *is* optimal to exercise immediately. Indeed it can be shown rigorously that the decision rule τ that achieves the maximum in (5) above is given by

$$\tau(v) = \begin{cases} 1 & \text{if } (S(v) - L)^+ = V(v) \text{ or } \delta(v) = N, \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

Let us look at the optimal exercise rule from a cost-benefit viewpoint. We can think of the strike price L as the "cost" of exercising the option, since this is the price one must pay to obtain a share of stock. Similarly, S_k is the benefit from exercising at time k, since this is the price one would obtain by selling the stock in the market. We just remarked above that it may not be optimal to exercise as soon as the benefit S_k exceeds the cost L. To explain this, it will be useful to view the option value V_k as representing the "value of the choice to exercise". When the option is exercised, the option (and the choice) is killed and this value is lost, so that V_k represents the *opportunity cost* of exercising the option. Thus when the option is exercised, there are two costs: the *direct* cost L, and the opportunity cost V_k. From the discussion above, the optimal exercise strategy is to exercise when $(S_k - L)^+ = V_k$, which in cost-benefit terms can be stated as: *Exercise only when the benefit S_k equals the direct cost L plus the opportunity cost V_k.* This viewpoint is the one that we will find most useful in this paper.

The value V_k can be computed for all k by a simple *dynamic programming* procedure (see [5]) as follows. First observe that $V_N = (S_N - L)^+$. This is clear both from formula (5) and from observing that the since option expires at time N, there is no advantage to waiting. Now stepping backward in time in the decision tree, we compute $V_k(v)$ at any depth k node v by

$$V_k(v) = \max\{(S_k(v) - L)^+, \mathbf{E}(V_{k+1}|v)/R\}. \tag{7}$$

In other words, the option value $V_k(v)$ at a depth k node v is the maximum of the immediate payoff $(S_k(v) - L)^+$ and the expected present value of the option value one time step ahead, given that we are at v. It can be shown that this backward-recursive formula for V_k and formula (5) are equivalent. And this is true regardless of the specific process that the stock price S_k follows.

3 Real Options and Software Decisions

An investor holding an American call option is faced with a decision problem: when to exercise his option. This situation is very similar to the problem faced by managers making irreversible capital investment decisions. Suppose the manager of a manufacturing firm is contemplating whether to invest in a large factory for making a new kind of widget.

The investment is irreversible, since the factory can only be used to make these widgets, and if market conditions turn out to be unfavorable, the firm cannot regain its lost investment. For simplicity let us say that the factory can be built instantaneously, at a cost L, and that it can produce widgets forever at zero cost. Once the factory is built, say at time k, the widgets can be sold at the prevailing market price for such widgets. The future profits from widget sales depend on how the market price evolves, which is uncertain. Let S_k be the expected value of these future profits, discounted to time k, under a suitable market price model, probability measure, and discounting factor. Thus S_k represents the value of the asset that the firm can acquire by exercising its option to invest L at time k. Alternatively, one can think of S_k as the "benefit" from making the investment at time k, and L as the cost of the investment. In this respect, S_k is analogous to the price of a stock underlying an American call option, and L represents the strike price. Clearly the firm will not invest in the factory if $S_k < L$.

On the other hand, should the firm invest simply because S_k exceeds L? The traditional Net Present Value (NPV) rule recommends investing in this situation. However, this rule can often be suboptimal since it treats the decision problem as a now-or-never proposition. That is, if there is no possibility of delaying the decision, then the rule is indeed reasonable. However, if the decision to invest can be postponed, then the NPV rule ignores the value inherent in waiting for better information before making the investment. The option viewpoint is the natural framework in which to quantify the worth of the flexibility of being able to choose between investing now and any future time. Thus the value V_k represents the "value of the option to invest", or the opportunity cost of investing, at time k. The reason we think of V_k as an opportunity cost is that when we exercise the option, we lose the opportunity of being able to decide when to invest. In analogy with the above rule for an American call option, the optimal rule for the firm is (given suitable definitions of S_k and V_k): *Invest if the asset value (or benefit) S_k exceeds the direct cost of the asset L plus the opportunity cost V_k.* This idea is at the heart of the theory of real options [7, 12].

The above approach applies to any investment situation where (a) there is an expenditure of limited resources, (b) there is uncertainty over the future profitability of the investment, (c) the decision to invest is irreversible, and (d) the decision can be delayed. As we argued before, many software engineering design decisions, and prototyping decisions in particular, satisfy these criteria. In general suppose a software engineer is contemplating when (if at all) to invest in a prototype. In terms of the variables introduced above, the direct cost L is the cost of implementing the prototype. The future profit stream from prototyping is uncertain, and may depend on several factors, such as changes in requirements, hardware, usage-patterns, etc. This uncertainty can be modeled in terms of a decision tree (see Section 2.1 for definitions of this and other terms). At any discrete time k, the *asset value* or *benefit* S_k is the expected value, discounted to time k, of the future (possibly negative) profit stream that would result if the prototype were already in place by time k. The option value V_k is then the opportunity cost of implementing the prototype – the value of the flexibility (which is lost when the prototype is implemented) of being able to decide when to implement the prototype. Just as in the capital investment scenario above, the optimal decision strategy for the software engineer is:

> *Invest in implementing the prototype when the benefit S_k is at least equal to the direct cost L plus the opportunity cost V_k.*

We have thus a rigorous way of quantifying when it is beneficial to delay our decision to invest in a prototype. In the next section we will present a model for the effect of prototypes. We show how to compute the optimal timing of both the prototyping decision *and* the resulting design-change decision in the same decision tree. The above simple rule will need to be modified when both the prototyping and design-change decisions are considered together.

4 Prototypes as state-predictors

When a software designer is contemplating a major, expensive design change, he or she may be concerned about the wisdom of going ahead with the change, since the benefits of making the change may depend on uncertain future events, or "states of nature". In such a situation, as Boehm has argued, it may be possible to obtain information about the future states by building a *prototype* or *simulation* of the contemplated software design (or design change). Although we are modeling prototypes as revealing information about *future* states or events, essentially the same model applies to prototypes that are used to reveal information about the *current state*. In particular, we can view the prototype as being done "just before", the current state unfolds, and the same mathematics holds. For instance in Boehm's [3] example the prototype reveals information about whether or not the current state is favorable to a certain design change. Our models are more general and cover scenarios such as those discussed by Boehm. To understand our model it will be useful to review Boehm's example in [2] which we describe in the next subsection.

4.1 Boehm's example

In the example in Boehm's book [2], we are required to choose between a bold (B) and conservative (C) approach in developing a certain special-purpose operating system. Since there is no operating system to start with, a choice between these two approaches must be made at the present time, which we call time 0. However, the profitability of the operating system will depend on the "state of nature" at time 1. There are two possible states of nature at time 1:

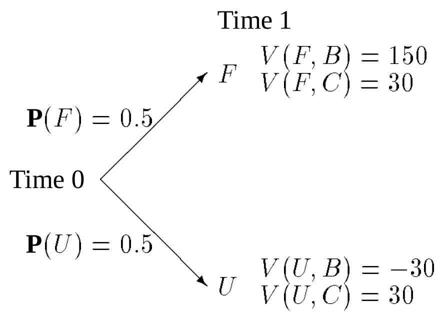

Figure 1: *Payoffs in Boehm's example*

"favorable" (F) and "unfavorable" (U), each occurring with probability 0.5 (see Fig. 1): $\mathbf{P}(U) = \mathbf{P}(F) = 0.5$. In general we denote the payoff of approach a in state s by $V(s, a)$. In the absence of any information about the state at time 1, we would choose either approach B or approach C at time 0. In the bold case, the expected payoff would be

$$P(F)V(F,B) - P(U)V(U,B) = 60,$$

and in the conservative case the expected payoff is 30. Can we do better if we have some idea of what the next state at time 1 will be? Boehm considers the idea of building a rough prototype of the key portions of the bold approach, in order to get information about the time-1 state. First consider a *perfect* prototype, namely one that will predict exactly what the next state will be. Let us say the cost of building this prototype is $C_p = 10$. If the prototype tells us that the next state will be favorable, we would decide on approach B, and otherwise we would pick approach C. The expected net payoff in this case would be

$$\mathbf{P}(F)V(F,B) + \mathbf{P}(U)V(U,C) - C_p = 90 - 10 = 80,$$

which is better than the payoffs without prototyping.

One way to view the effect of the perfect prototype in this example is that it gives us a way of making our decision at time 0 be *contingent* on the next state. To see this, we let $\mathbf{P}(a|s)$ denote the conditional probability that we pick approach a at time 0 given that the *next* state is s. In the absence of a prototype, our decision cannot be contingent on the next state, so we would either have

$$\mathbf{P}(C|U) = \mathbf{P}(C|F) = 1,\ \mathbf{P}(B|U) = \mathbf{P}(B|F) = 0, \tag{8}$$

which corresponds to picking choice C, or

$$\mathbf{P}(C|U) = \mathbf{P}(C|F) = 0,\ \mathbf{P}(B|U) = \mathbf{P}(B|F) = 1, \tag{9}$$

which corresponds to picking choice B. Now if the perfect prototype predicts that state F will occur at time 1, we would pick approach B at time 0, otherwise we would pick approach C (i.e. our decision is contingent on the *next* state), so that

$$\mathbf{P}(C|F) = 0,\ \mathbf{P}(C|U) = 1,\ \mathbf{P}(B|F) = 1,\ \mathbf{P}(B|U) = 0.$$

Note that with or without a prototype we have

$$\mathbf{P}(B|F) + \mathbf{P}(C|F) = 1,\ \mathbf{P}(B|U) + \mathbf{P}(C|U) = 1. \tag{10}$$

Thus our expected payoff with the perfect prototype can be written

$$\begin{aligned}&\mathbf{P}(U)\left[\mathbf{P}(B|U)V(U,B) + \mathbf{P}(C|U)V(U,C)\right]\\ &+\ \mathbf{P}(F)\left[\mathbf{P}(B|F)V(F,B) + \mathbf{P}(C|F)V(F,C)\right] - C_p,\end{aligned} \tag{11}$$

which is 80, as computed before.

We can consider more generally an *imperfect prototype,* one where the only restriction is that the conditional probabilities $\mathbf{P}(a|s)$ satisfy (10). The expected net payoff from the prototype is still given by (11). In particular, *not* using a prototype can be viewed as using a completely uninformative prototype, in which case the conditional probabilities would be given by either (8) or (9). Let us consider the prototype (costing 10) in Boehm's example, characterized by the conditional probabilities

$$\mathbf{P}(C|F) = 0.1,\ \mathbf{P}(C|U) = 0.8,\ \mathbf{P}(B|F) = 0.9,\ \mathbf{P}(B|U) = 0.2. \tag{12}$$

Note that condition (10) is satisfied. Now the expected payoff, from (11), is 68.

Thus the effect of any prototype can be modeled by the conditional probabilities of choosing various alternatives, in each of the possible *next-states.* We do not make explicit how these conditional probabilities are arrived at.

4.2 Our model

As we just saw above, Boehm's example (as well as his generalization) considers only one time-period. Moreover, the prototyping decision and the choice between bold/conservative must be done at the present time, since there is no software to start with. We now extend this model in two ways: (a) We consider multiple time-periods, in the event-tree framework. (b) We also assume that a software design is already in place, and a *design-change* is being considered. This means that both the prototyping and design-change decisions can be delayed. In particular the implementation-change does not have to take place immediately after prototyping – the software engineer can wait until conditions (the "state of the world") are appropriate for this. We first present a model for changing to a specific new design, and in subsection 4.5 we generalize this to switching to one of several alternatives. We refer to the design-change decision as the *switching decision*. The ability to delay both the prototyping and switching decisions can lead to increased payoffs for the software engineer.

We will henceforth use the word "prototype" to cover simulation as well. Although less expensive than changing the software, prototyping can still have a significant cost. Moreover, the expense incurred in prototyping cannot be recovered, [1] making a prototyping decision an irreversible one. However, the designer has the option to *wait* for better information before building a prototype.

We present our model through a concrete example that is easy to generalize. Suppose a database company X is contemplating whether to change the cache-coherence protocol on their top-of-the-line distributed DBMS. Since the future is unknown, it is unclear how this change will impact the company. It may be that in some scenarios the usage of the DBMS is such that the change is unfavorable. The company therefore wants to invest in prototyping and simulation. It is faced with two decisions:

- Decision P: When (if at all) to invest in prototypes and simulation software? For simplicity, we assume that that prototypes and simulation software are built together and instantaneously. To allow for different levels of prototyping, we assume that the set $A = \{a_1, a_2, \ldots, a_r\}$ represents the possible amounts of money that can be invested in prototyping at any time.
- Decision D: when (if at all) to switch to the new cache-coherence protocol? We of course impose the restriction that prototyping can not occur after switching to the new protocol.

We formulate our model in terms of a decision tree of depth N (See Section 2.1 for basic mathematical definitions). The branches of this decision tree represent various future events that might have an impact on how much benefit the new cache coherence protocol would bring. Throughout this discussion we will refer to the depth 3 decision tree in Figure 2 for illustration purposes.

In general, we assume that at any time there are c possible *levels* of prototyping (i.e., the possible amounts of money we could spend prototyping), which we denote by $\{1, 2, \ldots, c\}$. A specific prototyping strategy can then be described by a decision rule (as defined in Section 2.1) π that maps nodes v in the decision tree to an element of $\{0, 1, 2, \ldots, c\}$. As before, we interpret $\pi(v) = 0$ as not investing in a prototype at node v. We let τ denote the decision rule that describes the timing of the decision to invest in the new cache-coherence protocol. We assume that there is a fixed cost to switching to the new protocol, and therefore let τ be a map from the nodes to $\{0, 1\}$. Since prototyping (if any) can never occur after

[1] Unless the prototype can be a part of the system being built. In the case of simulations the expense is always unrecoverable.

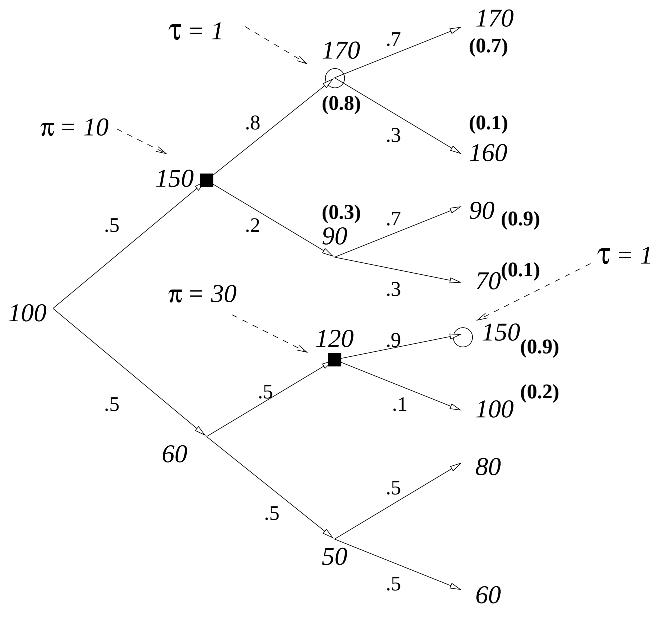

Figure 2: A decision tree showing a prototyping strategy π and a strategy τ for switching to the new implementation. The dark boxes indicate nodes v where $\pi(v) > 0$, and the circles indicate nodes where $\tau(v) = 1$. The benefit S_k is shown alongside each node in italics. The conditional probabilities $\mathbf{P}_k^\pi$ are shown in parentheses at the nodes where they are well-defined. In the paper, a node is referred to by the sequence of up (U) and down (D) branches that must be taken to reach the node. E.g., at node UD, the benefit $S_2 = 90$.

switching to the new protocol, we impose the restriction, for any node w:

$$\text{If } \pi(w) > 0, \text{ then for all } v \rightarrow w,\ \tau(v) = 0. \qquad (13)$$

Note that we are allowing paths where prototyping is never undertaken (i.e. all nodes v on the path have $\pi(v) = 0$) but the new cache-coherence protocol is implemented ($\tau(v) = 1$ somewhere on the path). Thus for a given prototyping decision π, there is a set $\mathcal{S}(\pi)$ of decision rules τ for decision D that satisfy the constraint (13). See Fig. 2 for an example of π and τ that satisfy this restriction.

After doing a certain level of prototyping at a node v, we can observe our path on the decision tree, and make the switching decision at some node w (say at depth k) that is a descendant of v. When making this decision, we are using the prototype to give us some indication of what the *next* (i.e. time-$(k + 1)$) state will be. This indication may be in the form of an explicit state-prediction, but not necessarily so. In any case, as in the numerical example of the previous subsection, we abstractly model the effect of the prototype as follows. For each time-$(k + 1)$ state α such that $w \rightarrow \alpha$, we have a *conditional probability* of having switched to the new implementation at node w given that the successor-state of w is α. We denote this conditional probability by the random variable $\mathbf{P}_{k+1}^\pi(\alpha)$, where we are using the superscript

π to emphasize the dependence on the prototyping strategy π. Note that the random variable $\mathbf{P}^\pi$ is only well-defined on nodes α that are descendants of nodes v where a non-zero prototyping investment is made, i.e., $\pi(v) > 0$. This does not cause a problem since we will only use $\mathbf{P}^\pi$ where it is defined. For instance in Fig. 2, the conditional probability $\mathbf{P}_1^\pi$ at node U is not defined since there is no prototype at time 0. On the other hand, at node UD we have $\mathbf{P}_2^\pi = 0.3$, which means the conditional probability of having switched to the new design at time 1 at node U, given that the state at time 2 is UD, is 0.3.

For a node v at depth k, $S_k(v)$ will denote the **benefit** at time k, at node v, from the new implementation, if the switching decision was made at time $k-1$ at the parent node of v. In general S_k represents the present (i.e. time-k) value of all future benefits from the new implementation. We let L denote the **direct cost** of switching. We will assume a discount factor of R per time period.

Note that if we make a switching decision at some depth k node w and there is no non-zero investment in prototyping prior to w on the path to w, then we immediately get a payoff $(S(w) - L)^+$, which we denote by $G_k^\pi(w)$. This is also true if we switch to the new implementation at a leaf (depth N) node in the tree, since there is no uncertainty after time N. On the other hand, if w is not a leaf node, and there *was* some non-zero level of prototyping at an ancestor of w, then even though we speak of a "switching decision" at time k (at node w), we are not necessarily switching since, depending on the probabilities $\mathbf{P}_{k+1}^\pi$, there is some chance that we do not switch. In fact we are deciding whether to switch or not to switch, and if we happen to decide *not* to switch, we are committed to this decision and cannot undo it subsequently. Thus the switching decision made at w translates into a probabilistic outcome at the successor-nodes of w. Therefore the effective benefit at time k is the expected benefit from time $k+1$ (weighted by the conditional probabilities $\mathbf{P}_{k+1}^\pi$) discounted to time k, given that we are at w, which is $\mathbf{E}(S_{k+1}\mathbf{P}_{k+1}^\pi/R|v)$. Therefore the **payoff** $G_k^\pi(w)$ at such a node w is

$$G_k^\pi(w) = \left[\mathbf{E}\left(S_{k+1}\mathbf{P}_{k+1}^\pi/R\,\middle|\,w\right) - L\right]^+, \qquad (14)$$

From the above description it is clear that the effect of a prototyping decision π is captured in the payoff process $\{G_k^\pi\}$. Thus at any time k, for any allowable prototyping strategy π and any allowable switching strategy $\tau \in S(\pi)$, the value of these strategies at time k, given that we are in state v, is (in analogy with (4) in Section 2.2)

$$V_k^{\pi,\tau}(v) = R^k\,\mathbf{E}\left[\frac{\tau G^\pi}{R^\delta} - \frac{\pi}{R^\delta}\,\middle|\,v\right]. \qquad (15)$$

Therefore, in analogy with definition (5), we can define the value of our investment opportunity at a depth k node v as

$$V_k(v) = \max_\pi \max_{\tau\in\mathcal{S}(\pi)} V_k^{\pi,\tau}(v). \qquad (16)$$

Since the number of possible strategies π and τ is finite, both $\max$'es above exist. A modification of the dynamic programming algorithm of Section 2.2 can be used to compute V_0, as well as the optimal times π^* and τ^* for decisions P and D respectively.

A reasonable question at this point is to ask, "what do the optimal strategies π^* and τ^* look like"? For instance, the optimal exercise strategy in the case of a simple call option is easy to characterize (see Section) 2.2):*Exercise when the option value V_k equals the payoff from immediate exercise.* Is there an analogous rule when we are considering two strategies π^* and τ^*? To answer this, we must first define what "the payoff from deciding to prototype" means (Note that (14) already defines the payoff from a switching decision). Suppose we decide to invest at level x in a prototype at a depth k node v, i.e., $\pi(v) = x$. What is the payoff from this decision? This is simply the maximum discounted expected payoff given that we are at node v, over all possible switching strategies τ:

$$Y_k^\pi(v) = \max_\tau \mathbf{E}\left[\tau G^\pi R^{k-\delta}\,\middle|\,v\right] - \pi(v). \qquad (17)$$

Note that this payoff itself is the value of an option, so in this sense when we are considering the two strategies π and τ, we have an *option on an option.* Another point to note is that the first term in the payoff represents the "benefit" from prototyping at time k at level x, whereas the second term, x, of course represents the cost of prototyping. In analogy with our characterization of the optimal exercise strategy for a simple call option, we can say (and this can be proved rigorously) that the optimal prototyping strategy is to prototype at time $k \le N$ at a node v at level x if and only if the payoff $Y_k^\pi(v)$ equals the option value $V_k(v)$. As mentioned before, we can view V_k as the "opportunity cost" of prototyping at time k since this represents the option value that is lost when we decide to prototype at time k. Therefore in analogy with call options, the optimal prototyping strategy is:

> *Implement a prototype at time k at level x only if the benefit equals the prototyping cost x plus the opportunity cost V_k.*

To re-emphasize a point we have been making throughout this paper, previous work on prototyping that was based on the traditional NPV rule has ignored the opportunity cost V_k. The above rule shows that it may be better to postpone prototyping even when the traditional NPV rule recommends it.

Now, once prototyping has been done at some level $\pi(v)$ at a node v, we must still decide when to switch to the new implementation. This consists of finding which τ maximizes the payoff $Y_k^\pi(x)$ defined in 17. At this point the problem is very similar to that of deciding the optimal exercise strategy of a call option, and this analogy has been explored at great length in our earlier paper [14]. We only note here that it may be optimal to wait for a while after prototyping before switching to the new implementation.

4.3 Boehm's example revisited

As an illustration, we now analyze Boehm's one-period example of Subsection 4.1 in the model just introduced. We will interpret that example from the viewpoint of switching from the conservative to the bold approach. That is, we assume that the conservative approach (C) is already in place, and we are required to decide whether to switch to approach B. Our decision tree is simple: the time-0 state has two successor states at time 1: favorable(F) and unfavorable (U). We want to define $S_1(v)$, the *additional* benefit (compared to the conservative approach) that would be realized in a time-1 state (or node) v, by switching to the bold approach at time 0. Thus, in the favorable state F, $S_1(F) = 150 - 30 = 120$, and in the unfavorable state U, $S_1(F) = -30 - 30 = -60$. The direct cost L of switching is 0. Note that Boehm did not consider the cost of switching. In this simple decision tree, we cannot delay the prototyping and switching decision, i.e., we must make them at time 0. We will take the discounting factor R to be 1.

Consider the imperfect prototype at the end of Subsection 4.1, whose conditional probabilities are given by (12). The cost of this prototype is 10, which we view as "level 10". Since we are using this prototype at time 0, $\pi = 10$ at the root of the tree, and $\pi = 0$ at the two depth 1 nodes. The effect of this prototype is characterized by the conditional probabilities $\mathbf{P}_1^\pi = \mathbf{P}(B|F) = 0.9$ in state F and $\mathbf{P}_1^\pi = \mathbf{P}(B|U) = 0.2$ in state U. Given that we have used the prototyping strategy π, the payoff G_0^π from making the switching decision at time 0 is, from the definition (14):

$$G_0^\pi = \mathbf{E}(S_1\mathbf{P}_1^\pi/R) - L = (0.5)(120)(0.9) - (0.5)(60)(0.2) = 48.$$

The net payoff, taking into account the cost of prototyping is, from (15): $\mathbf{E}\left[G_0^\pi/R^0 - \pi/R^\delta\right] = 48 - 10 = 38$. This is the net *additional* payoff of switching to the bold approach, compared to the conservative approach. Since the payoff of the conservative approach is 30, the actual payoff from choosing an implementation at time 0 based on the prototype is $30 + 38 = 68$, which matches the value obtained at the end of Section 4.1.

4.4 Another example

We illustrate the benefit of using our options approach by carrying out some computations in the decision tree shown in Fig 2. For this example, we assume the cost L of switching to the new software design is 100. We will assume a discount factor $R = 1.1$.

In the figure we show a specific prototyping strategy described by the decision rule π, and a specific switching strategy described by the decision rule τ. For instance, at the node U, $\pi = 10$, meaning that prototyping is done at this node at "level 10". Similarly, at node UU, $\tau = 1$, meaning that the switching decision must occur at this node. It is important to keep in mind that the benefit $S_k(v)$ shown at each node v is the present (i.e. time-k) value of all future profits that would be realized, *if* we switch to the new implementation at the parent-node of v, *and* state v occurs. Note that there is no (non-zero) prototyping on paths starting with DD. The switching strategy τ satisfies the constraint mentioned in the previous subsection: on paths where prototyping is done, the switching cannot occur before prototyping. Recall that we have modeled prototyping as inducing conditional probabilities $\mathbf{P}_k^\pi$ at the tree nodes.

We now compute the value of the strategies π and τ at time 0, namely $V_0^{\pi,\tau}$, as given by the expression (15). First we compute the payoffs G_k^π at nodes where a switching decision is made, i.e. UU and DUU:

$$G_2^\pi(UU) = \left(\frac{170(0.7) + 160(0.1)}{R} - 100\right)^+ = 22.27,$$

$$G_3^\pi(DUU) = 150 - 100 = 50.$$

The expected payoff from the switching strategy τ is

$$\mathbf{E}(\tau G^\pi/R^\delta) = \frac{(0.5)(0.8)22.27}{(1.1)^2} + \frac{50}{(1.1)^3} = 15.81.$$

And the expected cost of the prototyping strategy π is

$$\mathbf{E}\left(\frac{\pi}{R^\delta}\right) = \frac{(0.5)(10)}{1.1} + \frac{(0.5)^2 30}{(1.1)^2} = 10.73.$$

Therefore the value at time 0 of these strategies is $15.81 - 10.73 = 5.08$.

Now suppose we use the traditional NPV approach to decide whether or not to prototype (at level 10) *and* switch (at cost 100) at time 0, for a total cost of 110. Suppose that the prototype results in a conditional probability of 0.7 of switching in state U (benefit 150), and probability 0.4 in state D (benefit 60). The expected discounted benefit is therefore $((0.5)(0.7)150 + (0.5)(0.4)60)/1.1 = 58.6$, so the net payoff is $58.6 - 110 = -41.4$ units. The NPV approach would therefore recommend not investing in prototyping and switching to the new implementation. However we just saw above that by using the strategies π and τ we can get a positive expected payoff. This example illustrates a general principle:

> *When future benefits are uncertain, it may be profitable to delay the prototyping (or implementation-change) decision until the value from prototyping (or changing the implementation) becomes large enough.*

4.5 Switching to one of several alternatives

In the previous subsections we have assumed that there is some software implementation already in place, and we considered when (if at all) to switch to a specific different implementation. We now show how this can be easily generalized to the case of switching to one of m possible alternatives. Recall that $\mathbf{P}_k^\pi(w)$ for a depth k node v was defined as the conditional probability of having switched to the new implementation at the parent node v of w, given that the successor of v is w. Generalizing this, we define $\mathbf{P}_k^{i,\pi}(w)$ to be the conditional probability of having switched to the i'th alternative at w's parent v, given that the successor of v is w. Next, recall that $G_k^\pi(v)$ was defined as the payoff from making a switching decision at depth k node v, and was calculated as the expected payoff from time $k+1$, given that we are at v. When there are m alternatives to choose from, we should generalize this to be the *maximum* of the quantity analogous to $G_k^\pi(v)$ over all alternatives:

$$G_k^\pi(v) = \max_{1\le i\le m} \left[\mathbf{E}\left(S_{k+1}\mathbf{P}_{k+1}^{i,\pi}/R|v \right) - L \right]^+ . \tag{18}$$

With these definitions, the time k value of a specific prototyping and switching strategy is given by (15), and the time k value of our investment opportunity is given by equation (16).

5 Conclusion

We began this paper by pointing out the limitations of previous economics approaches to prototyping decisions, focusing in particular on the Bayesian approach. This model recognizes that prototyping can involve significant expenditure, and that the benefits from prototyping depend on uncertain future events. It uses the classical economics approach to decision-making under uncertainty, namely Net Present Value (NPV) analysis: invest in prototyping only if the benefits (on average) outweigh the costs. One problem with this approach is that it views prototyping decisions and implementation decisions as occurring simultaneously. It is more realistic to separate these decisions in time. Moreover, this framework does not consider the possibility of *delaying* the prototyping (or implementation) decision until better information becomes available. As a result, this approach may lead to sub-optimal prototyping decisions.

In this paper, we presented a model that addresses both these limitations: prototyping and implementation decisions that depend on prototyping are separated, and can occur at any time. In allowing the decisions to occur at any time, we can no longer use the classical NPV analysis. We must instead appeal to a more advanced but well-established theory, namely that of financial options. The basic ideas of this approach to software decisions were introduced by the present authors in a different paper [14]. In a sense the present paper extends that approach to software prototyping decisions. However, prototyping adds two novel twists: Firstly, the prototyping decision can change the future payoffs (depending on which of the two models we use). Secondly, we are now considering *two* decisions (prototyping and design-change) rather than just one. The models presented in the present paper are therefore significantly richer.

The combination of prototyping and design-change decisions is a special kind of *sequential investment* scenario studied in financial contexts [7], since the prototyping cost and the cost of switching to a different implementation are, after all, investments with an uncertain profitability. A firm may invest in a large factory stage by stage, and may want to time its decisions in order maximize the expected payoff. In a future paper we plan to further explore the connection between sequential investment and prototyping.

REFERENCES

[1] C. Baldwin and K. Clark. Modularity-in-design: An analysis based on the theory of real options. *Management Science*, 1994.

[2] B. W. Boehm. *Software Engineering Economics*. Advances in Computing Science and Technology. Prentice Hall, 1981.

[3] B. W. Boehm. Software engineering economics. In *IEEE Transactions on Software Engineering (SE)*, volume 10, Jan 1984.

[4] R. Brealy and S. Myers. *Principles of Corporate Finance*. McGraw-Hill, 1992.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *The Design and Analysis of Computer Algorithms*. Cambridge, Mass.: MIT Press, 1990.

[6] G. Daily. Beyond puts and calls: Option pricing as a powerful tool in decision-making. *The Actuary*, 30(3), 1996.

[7] A. Dixit and R. Pindyck. *Investment under uncertainty*. Princeton Univ. Press, 1994.

[8] A. Dixit and R. Pindyck. The options approach to capital invesment. *Harvard Business Review*, pages 105–115, May-June 1995.

[9] J. Flatto. *The application of real options to the information technology valuation process: A benchmark study*. PhD thesis, Univ. of New Haven, 1996.

[10] J. Hull. *Options, Futures, and Other Derivative Securities*. Prentice Hall, 2nd edition, 1993.

[11] J.Sick. Real options. In R. Jarrow, V. Maksimovic, and W.T.Ziemba, editors, *Finance*, volume 9 of *Handbooks in OR and Mgmt. Sc.* Elsevier/North-Holland, 1995.

[12] R. McDonald and D. Siegel. The value of waiting to invest. *Quarterly Journal of Economics*, 101:707–727, 1986.

[13] K. Sullivan. Software design: the options approach. In *Proc. Software Architectures Workshop*, 1996.

[14] K. Sullivan, P. Chalasani, and S. Jha. The options approach to software design. Technical report, University of Virginia, Computer Science Dept., 1997. Also submitted to IEEE Trans. Soft. Engg.

[15] J. Withey. Investment analysis of software assets for product lines. Technical Report CMU/SEI-96-TR-010, Carnegie Mellon University – Software Engineering Institute, Nov. 1996.