A Design-based Model for the Reduction of Software Cycle Time

Ken W. Collier, Ph.D.
Assistant Professor
Computer Science and Engineering
Northern Arizona University, Flagstaff, AZ
Ken.Collier@nau.edu

James S. Collofello, Ph.D.
Professor
Computer Science
Arizona State University, Tempe, AZ
James.Collofello@Asu.Edu

Abstract

This paper presents a design-based software cycle time reduction model that can be easily implemented without replacement of existing development paradigms or design methodologies. The research results suggest that there are many cycle-time factors that are influenced by design decisions. If manipulated carefully it appears that an organization can reduce cycle-time and improve quality simultaneously. The preliminary results look promising and it is expected that further experimentation will support the use of this model. This paper will analyze the basis for the model proposed here, describe the model's details, and summarize the preliminary results of the model.

Introduction

Many organizations have relatively mature, effective software-development processes in place and have employed talented software engineers and managers to implement these processes. In such organizations, it is appropriate to question whether each software-development effort is performed at peak efficiency and effectiveness.

One of the primary goals of any organization is to release high quality software in as little time as possible. Therefore, an increase in software-development efficiency may be manifest as a reduction in development time. However, simply reducing development time is not good enough. It is important that these reduction efforts maintain high levels of both process and product quality.

This research involves the examination and control of factors that have an impact on software development time. More specifically, this paper presents a model for reducing the development cycle time by restructuring software design. Reduction of development time, together with increasing quality and productivity, is the goal of many software development organizations. The benefits are numerous including: extended market life; increased market share; higher profit margins; and the ability to take advantage of emerging technologies [1]. Although much has been written about cost estimation, effort estimation and productivity and process improvement, little research has been directly aimed at reducing software cycle time.

The proposed model is based on a systematic and rigorous analysis of the software process and identification of product factors that affect cycle-time, as well as the impact of design decisions on those factors. This design-based model attempts to reduce software development time by iteratively and strategically restructuring the design of a software system. In order to be practical, such a model must provide significant benefits at a minimal cost. It must be able to be implemented simply without requiring an excess of special training. It must be applicable to as many different types of software systems as possible, and it must provide measurable benefits. It should also work in concert with, instead of as a replacement for, existing models and methodologies.

This paper is divided into three major sections. The first section provides motivation for the design-based model presented in the paper. It does so by demonstrating the impact of software design decisions on cycle-time, software quality, and scheduling options. The second section details the model and provides theoretical justification for an algorithmic approach to schedule refinement. The model also provides some guidance for restructuring the design to further shorten cycle time. Finally, the results of an evaluation of the model are provided. Although these results are inconclusive, they suggest that the model has promise and meets the requirements previously established.

Software Cycle Time Factors

Design Decisions Affect Cycle Time and Software Quality

Early in this research project, a systematic analysis of cycle-time factors and issues was conducted. The details of this analysis are outlined in [2]. However, an overview of

1060-3425/96 $5.00 © 1996 IEEE

the results of that study will serve to motivate the design-based model proposed in this paper.

First, the software development cycle was examined to establish a comprehensive set of factors impacting cycle-time. These factors were divided into process factors and product factors. Process factors include: software reuse, requirements change, risk management, productivity, personnel availability, software tools, equipment resources, method maturity, verification and validation techniques, quality assurance techniques, integration strategies, and integration schedule. Product factors include: subsystem dependencies, module reuse, subsystem complexity, subsystem independence, subsystem risk, subsystem size, and subsystem generalization.

Notably, most of these factors are impacted by design decisions. This recognition led to a set of cycle-time improving design goals including: maximize reuse; design independent subsystems; design simple subsystems; ensure completeness; localize risk; design to minimize risk occurrence; ensure correct design; design for low coupling; design for high cohesion; design to maximize independence; design unique subsystems; and design for flexible scheduling.

A fortunate, and unexpected side-effect of this analysis is that these are the same design goals that improve design quality. This analysis showed that design decisions impact both design quality and cycle-time, and that these goals are not mutually exclusive. Moreover, the analysis revealed that improving cycle-time through design decisions is likely to improve quality and vice versa.

Design Decisions Impact Development Schedule

Scheduling decisions play a critical role in reducing cycle-time. Therefore, it is appropriate to analyze the factors that constrain scheduling options. The architectural design of a system dictates subsystem dependencies, thereby dictating the possible scheduling strategies. Prior to software design, implementation scheduling possibilities are relatively unbounded by product decisions, i.e., all schedules are accessible.

Selecting a design alternative can be viewed as a reduction in the set of accessible schedules [2]. There are many decisions that further constrain the set of accessible schedules. Staffing, resource allocation, risk analysis, and identifying high-priority functionality, all constrain the set of accessible schedules. Therefore, making design decisions that home in on the optimal schedule is a cycle-time reduction goal.

It is unrealistic to suggest that one can always arrive at the best possible design or schedule. Moreover, there is no way of knowing when the best design or schedule has been developed. A more practical goal is to make design and scheduling decisions that increase the chance that a good schedule will be found. The software design and scheduling process can be viewed as a process of making decisions to maximize the chance of finding the best accessible schedule, given all scheduling constraints [2].

Currently, this attempt at finding a best accessible schedule is a function of intelligent decision-making, intuition, and the application of design and scheduling heuristics. An approach that is highly subject to the abilities and expertise of the software engineers and project managers. An ideal scheduling methodology should be a prescriptive process that encourages the result of a good schedule regardless of expertise and ability.

An Overview of Project Scheduling

In the late 1950s two computer-based scheduling systems were developed to aid in the scheduling of large engineering projects. Both are based on task dependency networks. The critical-path method (CPM) was developed by Du Pont and Remington Rand. The CPM method is a deterministic scheduling strategy that is based on the best estimate of task-completion time. The program evaluation and review technique (PERT) is a similar method and was developed for the US Navy. The PERT method used probabilistic time estimates [3].

In the CPM approach, a task-dependency network is a directed acyclic graph (DAG), which is developed from two basic elements: activities (tasks) and events (completion of tasks). An activity cannot be started until its tail event has been reached due to the completion of previous activities. An event has not been reached until all activities leading to it have been completed.

Associated with each activity is an estimated time to completion. The critical path is the path through the DAG that represents the longest time to project completion. To help calculate critical path and identify critical activities a set of parameters is established for each activity including: duration, earliest start time, latest start time, earliest finish time, latest finish time, and float. Float refers to the slack time between earliest start and latest start. All activities on the critical path have a total float of 0. This reflects the idea that critical-path activities must be completed on time in order to keep the project on schedule [3].

PERT scheduling uses many of the same ideas as CPM. However, instead of task completion being a "most likely" estimate, it is a probabilistic estimate. The estimator predicts an optimistic estimate, o, and a pessimistic estimate, p, of task completion. The most likely time, m, falls somewhere between those values. The time estimates are assumed to follow a beta distribution. The expected time is given as $t_e = (o + 4m + p)/6$. The expected times are calculated for each activity in the task-dependency network, and the critical path is then determined as in CPM [3].

Shortening the Critical Path

Software development time is measured by the critical path through the project's task dependency network. Hence, all cycle-time reduction efforts can be viewed as efforts to shorten the critical path. There are basically two ways to shorten the critical path: Shorten the completion time of tasks on the critical path; or, remove activities from the critical path

The first option can be achieved either by simplifying the task or by increasing the productivity of the work team assigned to the task. Task simplification can be accomplished by either dividing the task into simpler concurrent tasks or by eliminating unnecessary work from the task. Productivity can be increased by improving: management techniques, resources, or development techniques; or by allocating additional resources [4,5,6]. Although it is not a primary topic of this paper, the maximization of productivity is fundamental to cycle-time reduction.

Violating Task Dependencies

The second approach to eliminating critical-path activities implies the violation of task dependencies. This goal requires an understanding of the types of relationships between task dependencies. Tasks can be divided into two general subcategories: programming tasks and nonprogramming tasks (e.g., training, technical reviews, etc.). Programming tasks correspond to design components in a software system, and may be low-level modules, or the integrations of multiple modules into subsystems. Therefore, dependencies between programming tasks are connected to dependencies between the components in a design. There are three types of dependencies between tasks:

1. *Data Dependency* - If module A imports information that module B exports, then module A is data dependent upon module B.
2. *Functional Dependency* - If module A requires functionality provided by module B to complete its own function, then A is functionally dependent upon B.
3. *Resource Dependency* - If the completion of module A requires resources that are currently allocated to module B, then A is resource dependent upon B.

There is some cost involved in dependency violation. Otherwise, the ideal scheduling approach would be to complete all tasks in parallel and then integrate all at one time. Of course, as Fred Brooks observed, project scheduling and management is not this simple [7].

For programming tasks, the dependency violation cost occurs in the form of scaffolding (i.e., test drivers and code stubs) to simulate the parts of one module upon which another depends. As data and functional dependencies are violated, the amount of required scaffold development increases to simulate the dependency, thereby increasing the task completion time. This in turn increases the likelihood of defects being introduced into the code. As defects increase, the debugging time increases.

When resource dependencies are violated, the cost depends upon the type of resource. If two tasks require the use of some limited-access hardware device, then dependency violation might require the purchase of a second such device. The cost is monetary. If two tasks both require some very specialized expertise that few team members possess, then dependency violation means training other programmers. The cost in this case is in terms of time spent learning and retraining.

Many factors contribute to the completion time of programming tasks. Each programming task represents a cycle of subactivities that includes detailed design, coding, unit testing, integration testing, and system testing; all of which are time-consuming. Additionally, there are process-management and control activities involved in each task.

One must decide whether the benefits of violating a dependency outweigh the costs. Furthermore, it is unreasonable to expect that all dependencies share the same violation cost. The stronger or more complex the dependency, the greater the cost of violation. Dependency strength may be caused by the degree of coupling between modules or simply by the amount of access of the module's components by other modules.

Consider two tasks, A and B, in which B is dependent upon the completion of A. Dependency-violation cost can be viewed as the addition of some percentage of the completion time of A to B's completion time. This percentage reflects the amount of A that must be simulated in order to complete task B before A is actually complete. A violation cost approaching 100% reflects the idea that task A is being simulated in its entirety, which defeats the purpose of violating the dependency. Conversely, an estimated violation cost approaching 0% reflects the notion that B's dependency on A is artificial and both could be performed in parallel with little consequence.

A dependency classification scheme is proposed to help determine dependencies that are good candidates for violation. If a dependency violation cost is estimated to range between 0% and 25%, then the dependency is classified as a *weak* dependency. If the cost is in the range 26%-50%, then the dependency is *moderate*. If the cost is in the range 51%-75%, then the dependency is *strong*; while 76%-100% violation cost would be considered *very strong*.

Accurately estimating the cost of violating dependencies is a topic for future research. However, for programming tasks that are functionally or data dependent upon other tasks, the cost is primarily in the creation of code scaffolding. In this case, the estimated cost can be derived from the software cost models used to estimate the initial task durations. Resource dependencies and nonprogramming tasks are not likely to be so simple.

From these ideas on critical path shortening, a set of scheduling goals for the reduction of cycle-time form the basis for the model presented in this paper:

1. Violate low cost dependencies to increase concurrent development whenever cost effective.
2. Transform high cost dependencies into low cost dependencies.
3. Reduce task-completion time by simplifying the task.
4. Reduce task-completion time by dividing it into concurrent subtasks.
5. Reduce task-completion time by increasing productivity.

The Relationship Between Schedule and Design

The previous section was devoted to project scheduling and identifying the general strategies for shortening the critical path in a schedule. This section examines the impact of design decisions on scheduling outcomes. Moreover, identifying connections between design and scheduling provides a set of techniques for achieving the critical-path-shortening goals previously identified.

In general a software design and its development schedule are closely connected due to the fact that design components dictate the work tasks in the development schedule. It is useful to identify other, more subtle, design-schedule connections.

- *Design dependencies determine schedule dependencies* - Any degree of coupling between two modules in the design translates into a dependency between the corresponding tasks in the schedule.
- *Design independence determines task concurrency* - Modules in a design that are uncoupled can be developed in parallel, given no other constraints and assuming that they are not resource dependent.
- *Component complexity in design determines task-completion time* - Modules that are complex will take longer to implement and test than modules that are simple.
- *Component complexity in design determines task staffing* - Highly complex components may require additional personnel resources thereby limiting the degree of concurrency in the schedule.
- *Development learning curves affect productivity and productivity affects task completion time* - Design components that require programmers to learn special skills will take longer to implement than those components for which programmers are already trained.
- *Software reuse affects task-completion time* - Code reuse is almost certainly faster than designing, developing, and testing code from scratch.

In general, a design with complex components that are highly dependent upon one another will result in a highly serial schedule with long task-completion times and excess staffing needs. Conversely, a design with simple, independent components will result in a highly concurrent schedule with short task completion times and lower required staffing.

The schedule-improvement goals previously listed are impacted by *design decisions*. Task dependency strength is determined by *intermodular dependencies* (coupling). Task completion time is affected by *module complexity*. Task concurrency is affected by *intramodular dependencies* (cohesion).

Clearly, the coupling and cohesion heuristics play a role in determining the critical path length in a schedule. Additionally, information hiding, data abstraction, data localization, and fan-in/fan-out will affect the de

pendencies in a development schedule. This is further evidence that existing quality metrics are coincident with the goal of cycle-time reduction. Furthermore, these existing heuristics can provide the necessary mechanisms for improving the development cycle time by a judicious refinement of the design.

A Cycle Time Reduction Model

In [2] a link is made between the factors that affect

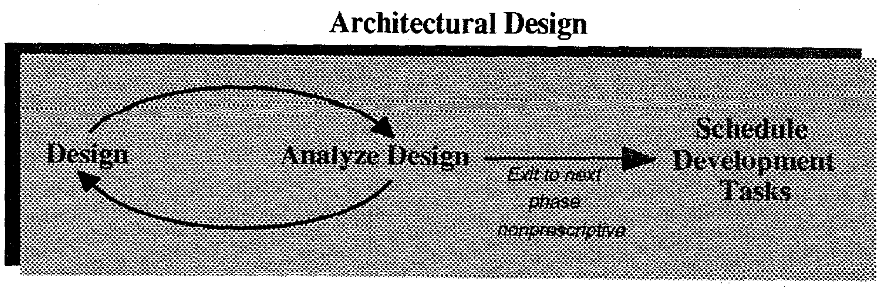
Architectural Design
Design
Analyze Design
Exit to next phase nonprescriptive
Schedule Development Tasks

Figure 1 - Current Design Cycle

development cycle time and software design decisions. This link combined with the link between design and scheduling provide sufficient motivation for a design-based cycle-time reduction model. The model proposed is based on the convergence of the ideas presented in previous sections. This model aims to shorten cycle time by shortening the implementation phase through design and schedule refinement.

Model Description

Current state of the art in software design dictates an iterative process of design and refinement to converge on a design of the highest quality. Figure 1 reflects this notion.

Under this model the development schedule is deferred until design is complete. There are some drawbacks to this design model. First, widely used design-based metrics do not provide a mechanism for determining when it is cost-effective to continue design refinement and when to stop refining and move to the next phase. Second, the current design model is not prescriptive. It is difficult to determine which parts of the design deserve refinement at any iteration of the cycle. Finally, although this iterative design process is the state of the art, it is not necessarily the state of the practice. It remains the tendency of many designers to adopt the first design that is generated. This may be because the benefits of iteratively improving design are difficult to quantify.

The model proposed in this paper is also iterative in nature. However, development scheduling becomes an integral part of the design cycle. In this model the designers develop an initial design and then iteratively work to develop the best schedule (i.e., shortest critical path) that can be implemented using that design. Then the design is refined according to standard practices. Following refinement, a new "best schedule" is developed for the improved design and so on until the schedule does not continue to improve. This model is graphically represented in figure 2.

The ultimate goal of this cycle-time reduction model is a design that improves development time without sacrificing product quality. This model addresses some of the problems with current iterative design models. Critical path length is the driving metric for determining when it is cost-effective to continue design refinement or when to move on to development. Furthermore, because critical path is used to reflect the improvement of each iterative design refinement, the benefits of the iterative process are measurable and can be compared directly to the cost of refinement. For example, if it takes five programmer days to conduct a single design iteration and the critical path is shortened by only three programmer days, then it is clearly not cost-effective to continue refining the design. In this case, the method helps designers determine when to stop designing. The approach taken by this model is to *selectively apply design-improving techniques to key components in the design.* The aim of this approach is to minimize the effort and maximize the benefit.

How The Design Model Works

This design cycle follows five basic phases:

1. *Initial Design* - During this phase the system is designed using existing methods and techniques. At this point in the process, the model does not differ from current design models. In fact, this cycle-time reduction design model may be thought of as a design metamodel, since it does not replace popular design techniques and paradigms but is symbiotic with existing methods.
2. *Initial Schedule* - The initial scheduling phase in this model employs current state-of-the-art scheduling and effort estimation techniques.
3. *Schedule Refinement* - The schedule-refinement phase of this design model is the point at which the schedule is iteratively analyzed and refined in an effort to shorten the critical path without altering the design.
4. *Problem Identification* - Now that the schedule has been refined sufficiently, this phase serves the purpose of identifying those system components that, if improved, represent a significant cycle-time improvement.

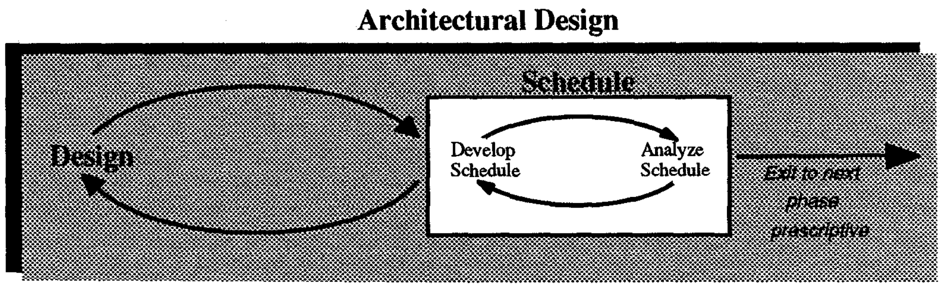

Figure 2 - Cycle Time Design Model

5. *Design Refinement* - During this phase efforts are made to improve those tasks that were identified as problematic by the previous phase.

This cycle is iterated using the critical path length as a cycle-time improvement metric. For each iteration of the design cycle, the critical path will reflect the benefit of the refinement efforts. Ideally, the architectural design cycle should halt as soon as the critical path becomes stable or when the benefits at each iteration do not justify the effort.

We do not propose to replace existing state-of-the-art design and scheduling methods in this model. Instead, the remainder of this section will focus on the development of methods for accomplishing steps 3, 4 and 5.

Schedule Refinement

The third phase in this model deserves closer inspection. Ideally, this phase will reveal the optimal schedule for the current version of the design. It has been observed that the task dependencies along the critical path in a schedule can, potentially, be violated. A brief foray into graph theory will motivate an algorithmic approach to the problem of schedule optimization.

The scheduling problem can be temporarily simplified by stating it as a graph-manipulation problem restricted by a set of rules. Given a directed acyclic graph (DAG) in which each vertex v has a weight that is represented by $wt(v)$, the **weight** of the entire graph W is:

$W = \Sigma\, wt(v_i)\, ;\exists c \forall v_i$ *{c is a critical path,* v_i *is a vertex /* v_i *is on c}*

The **burden** of a vertex v, $b(v)$ is the sum of the weights of all immediate predecessors of v, which are on a critical path. The **prehistory** of v, $ph(v)$ is the sum of the weights of all predecessors of v, which are on the longest serial path from the initial vertices to v. These are likely to be the predecessors of v that are on a critical path. The object is to reposition vertices in the graph in order to achieve the smallest weight (i.e., shortest critical path). However, there are rules for moving vertices:

1. Never disconnect the goal vertex from its predecessors. This is equivalent to eliminating subsystems from the final integration.
2. Vertices are repositioned by disconnecting them from their critical-path predecessors. This reflects the notion of violating task dependencies.
3. Whenever a vertex v is disconnected from its critical path predecessors, these predecessors are reconnected to all of v's immediate successors. This is to prevent the graph from becoming disconnected.
4. Whenever a vertex v is disconnected from its critical-path predecessors, its weight is adjusted by the expression: $wt(v) = wt(v) + (m \times b(v))$. In this expression m represents the percentage of v's dependent tasks that must be simulated in order to develop and test v (i.e., scaffolding). This reflects the cost of violating task dependencies.

Some observations can be made from this view of schedule optimization. First, it is not likely to be beneficial to reposition slack path vertices. Second, a path that is critical at one point in the optimization process may cease to be critical at some future time. Third, there may be multiple critical paths in the graph at any given time. These conditions and others that are not so obvious must be addressed by any strategy that is to be used to solve this problem.

An algorithmic strategy might implement a cycle of vertex repositioning followed by recalculating critical-path length. Such an algorithm must be able to determine which vertex to reposition at each iteration and when to stop.

The selection of a vertex for repositioning must be conducted in light of a cost-benefit analysis. The benefit is measured as a reduction in W. The cost is represented by the increase in $wt(v)$ due to dependency violation. The vertex that produces the greatest benefit and least cost is the prime candidate.

Some additional observations can be made about the structure of critical paths in this problem space. It is possible that the critical path in a DAG of significant size will not be a single linear sequence of vertices and edges. Multiple parallel critical paths may require repositioning multiple vertices before seeing a reduction in W. Furthermore, if the critical path diverges at one task and then reconverges later (i.e., becomes temporarily parallel), vertices prior to this divergence or following the reconvergence should be manipulated if possible in order to realize immediate benefits.

The trouble with each of these scenarios is that they obscure the benefits of repositioning certain candidate vertices. The value of W may remain steady over several iterations before it begins to decrease. Rather than halting when no further decrease in critical path length is seen, it may be more beneficial to halt when an increase is detected.

These observations motivate a simple greedy algorithm that iteratively repositions critical path vertices based on a cost-benefit analysis and recalculates the value of W, halting when the value of W increases. This algorithm always identifies the vertex that appears to promise the greatest immediate reduction in W. Greedy algorithms are often used in optimizing problems such as this. However, most greedy algorithms accept a "good" solution rather than guaranteeing an optimal solution. This algorithm is no exception. The problem of finding an optimal solution lies in the probability of reaching a local minimum It is possible that the repositioning of a vertex will temporarily increase W in order to realize a greater decrease on future iterations. Unfortunately, guaranteeing an optimal solution to graph-shortening for a graph of any complexity is a hard problem which cannot be solved in polynomial time and is NP-complete [8, 9, 10].

Calculating Dependency Violation Cost

In determining the net benefit of dependency violation, *m* represents the percentage of simulation of the tasks on which another task relies. This requires that up to 100% of the first task must be simulated in order to complete the second task. Therefore, the cost of violating a dependency lies between 0% and 100% of the duration of the first task. Hence, for a single dependency the value of *m* is in the range [0.0,1.0]. The value of *m* increases as the number of tasks on which the target task is dependent increases and may exceed 1.0.

Furthermore, whenever a task dependency is eliminated there are likely to be hidden costs due to added communication requirements, inaccuracies in estimating, etc. As *m* approaches 1.0 the likelihood increases that these hidden costs will cause the dependency violation cost to exceed the benefit. Therefore, it is useful to establish a maximum value for *m* above which a dependency should not be violated.

Empirically defining an upper bound value for *m* is extremely difficult if not impossible. However, during the development of the model presented here, fifty different schedules and scheduling scenarios were examined. The purpose of these examinations was to identify the behaviors and consequences of task dependency violation. These efforts led to the observation that .25 appears to be a conservatively reasonable upper boundary value for *m* [1].

It was consistently observed that whenever the estimated cost of violating a dependency was below 50%, the violation resulted in a decrease in critical path length. In fact, many critical path decreases were observed for cost estimates as high as 80%. However, providing for hidden costs, it is deemed prudent to take a highly conservative approach to selecting an upper bound value for *m*. It was observed during these exercises that 100% of the cases in which the value for *m* was below 30% resulted in a shortening of the critical path. Compensating for hidden costs, this model uses an upper bound of 25% for *m*. This choice has the added benefit of allowing the cycle-time reduction model to focus only on weak dependencies as candidates for violation. Assuming inaccuracies in the estimates of dependency violation cost, .25 represents a worst-case value within that range. Fixing *m* at .25 simplifies the algorithm since categorizing dependency strength is easier than accurately estimating the cost of dependency violation.

This use of .25 as a fixed value for *m* in the cycle-time reduction model can easily be replaced by a more empirical value or by the actual violation cost estimate. The selection of a fixed value serves primarily to simplify the use of the model. This value was selected under ultraconservative conditions and its use may result in a loss of accuracy. The remainder of the discussion of this model will use .25 as a value for *m*. However, until an empirically established upper bound value is established one may prefer to use actual violation cost estimates.

An Algorithm for Schedule Refinement

The graph theory concepts of the previous section form the basis for a schedule refinement algorithm that attempts to reposition critical path tasks. Both CPM and PERT represent a project schedule as a DAG in which vertices represent development activities and milestones, while edges represent dependencies between tasks. The duration of each task corresponds to the weight of a vertex, *wt(v)*, and the critical path duration corresponds to *W*. The prehistory of a task *t*, *ph(t)*, is the sum of the durations of all tasks preceding *t* along the critical path. The burden of a task *t*, *b(t)*, is the sum of the durations of all tasks that immediately precede t.

The process of refining the schedule is, in essence, a process of selectively violating task dependencies in a cost-effective manner. It is assumed that the schedule follows a typical CPM or PERT format, in which the information for each task includes: earliest start date (ES), latest start date (LS), earliest finish date (EF), latest finish date (LF), and duration (D). Furthermore, it is assumed that the dependencies between tasks in a schedule have been categorized using a scheme similar to the one previously described. The following strategy takes the conservative approach of violating only weak dependencies and assumes the worst case within that range (i.e., $m = .25$). Given more information, the procedure can be modified as is necessary.

The algorithm is as follows:

1. Build a set *D* of weak dependencies on the critical path. These are the primary candidates for violation.
2. For each dependency $i \rightarrow j \in D$, where task *i* is dependent on task *j*, calculate $ph(i)$ and $b(i)$. Remove from *D* any dependencies whose cost outweighs *i*'s prehistory, $wt(i) + ph(i) \leq wt(i) + .25b(i)$.
3. For each dependency $i \rightarrow j \in D$, calculate the net benefit, $n(i) = wt(i) - .25b(i)$, of removing the dependent task from the critical path and placing it in a slack path. Remove any dependencies from *D* that have no net benefit, $n(i) < 0$. *D* now contains the final set of candidates for elimination.
4. Select the dependency $i \rightarrow j \in D$, for which $n(i)$ is the greatest, remove the dependent task from the critical path, and replace it on a slack path using the following rules:
    - If *i* has no successors and *j* becomes disconnected from all successors, then do not violate this dependency since *j* will never become integrated

into the system without an additional integration step. In this case, *i* is the final system integration. This is not likely to be cost-effective.

- Connect *j* to all immediate successors of *i* and update *i*'s ES, LS, EF, and LF values. This reflects the change in the integration strategy for *i* and *j* (*i* can now be developed concurrently with *j*).
- Update *i*'s duration by a factor of .25, $D = .25D$
- All additional constraints that affect the development schedule should be maintained (e.g., resource allocation, risk prioritization, etc.). It is impossible to define rules for maintaining these constraints. Therefore, it is the responsibility of the scheduler to ensure that they are not violated.

5. Because a new critical path may emerge as a result of task repositioning, steps 1-4 are repeated until the schedule reaches a stable state (i.e., no changes can be made to the schedule) or until the critical path begins to increase.

At best, the result of this algorithm is an optimal schedule for the given design under the given constraints. At the least, the resulting schedule will not be any worse than the original schedule.

There are additional scheduling-improvement techniques that have not been addressed here, such as resource-leveling, which may help to shorten the critical path [3]. Used in conjunction with such techniques, the model described here offers a simple yet powerful means of reducing development time. It is based on observation, intuition, and the mathematical manipulation of task networks and CPM components. A major benefit to this scheduling method is that it is highly automatable once tasks have been identified and a preliminary schedule is developed.

Problem Identification

Once the development schedule has been refined there are two ways to further shorten the critical path: either remove tasks from the critical path; or shorten the completion time of tasks on the critical path. The only remaining critical tasks are those with high dependency-violation costs or weak dependencies for which there is no net benefit in removing them from the critical path.

This phase in the model attempts to shorten the duration of a target task; or to prescribe design refinements that will result in weaker dependencies. The approach taken is to identify the design component that, if refined will result in the greatest cycle-time benefit.

This approach is heuristic rather than algorithmic. General guidelines are as follows:

1. Identify tasks in the schedule that have the greatest impact on the critical path.
2. Identify dependencies that, if violated, would produce the greatest reduction in the critical-path length.
3. For the identified tasks, examine the potential for either simplifying or decomposing their corresponding design components and estimate the effort required to do so. Retain those that have the greatest benefit for the least effort.
4. For the identified dependencies, evaluate the effort required to weaken the dependency and evaluate the potential for success. Retain those that have the greatest benefit for the least effort.
5. From the retained tasks and dependencies, select the one that has the greatest overall net benefit in terms of critical-path shortening.
6. Ideally, one task or dependency will be refined on each design-cycle iteration. However, it may be that by performing simple refinements to several tasks and/or dependencies, a major shortening of the critical path will be experienced on one iteration. These decisions must be determined based on individual project scenarios.

Problem identification relies upon the talents of software engineers to make good decisions. These guidelines may easily be supplemented with additional, project-specific information to increase their benefit. Future work in validating cost and benefit estimating techniques will further strengthen this phase in the model.

Design Refinement

After identifying the most beneficial task or dependency for refinement, it is necessary to quickly identify the cause of the problem and resolve it. If the target of refinement is a dependency, then the aim is to weaken that dependency to make its dependent task a candidate for repositioning. Assuming the dependency is not resource dependent, it may be that efforts should be made to decrease coupling between corresponding design components using existing design heuristics and principles.

If the target of the refinement effort is task duration, reducing the complexity of corresponding components might be accomplished through simplification or subdivision. Subdivision implies that the module has poor cohesion. Ideally, system modules should be functionally cohesive. Therefore, the designer should work to reduce cohesion and divide the system component into multiple, independent subcomponents.

Preliminary Results

This model cannot guarantee a reduction in actual cycle-time since many things can happen between software design and product release. However, preliminary results suggest that the model has promise in achieving the

Num	Size Estimation Technique	Effort Estimation Technique	Scheduling Approach
1	Actual Size	Actual Effort	DPS
2	Actual Size	Actual Effort	Educated
3	Actual Size	COCOMO	DPS
4	Actual Size	COCOMO	Educated
5	Actual Size	Putnam	DPS
6	Actual Size	Putnam	Educated
7	Expected Size	COCOMO	DPS
8	Expected Size	COCOMO	Educated
9	Expected Size	Putnam	DPS
10	Expected Size	Putnam	Educated

Table 1: Demonstration Strategies

following results:

1. Significantly reduce the *estimated* development time of a software product at the design phase. For the purposes of this effort, any reduction of estimated development time of 5% or more will be considered significant.
2. Maintain or improve the quality of the initial design (i.e., the model will not reduce design quality).
3. Help to focus design-refinement efforts on beneficial design components.

Although empirical validation of this model is virtually impossible, demonstration of the model on a variety of software designs under a variety of conditions yields encouraging results. The model was applied to five software systems ranging in size from 736 source lines of code (SLOC) to 6,309 SLOC, and ranging in quality. Three factors were identified as affecting the initial schedule of a particular software design: size estimation technique; effort estimation technique; and scheduling technique. For size estimation in this analysis actual size and expected size were used. For effort estimation, actual efforts were used in addition to the COCOMO and Putnam models [11, 12]. In developing initial schedules, two approaches were used: dependency preserving scheduling (DPS) and educated scheduling. DPS refers to preserving all intermodular dependencies from the design on the corresponding development tasks. Educated scheduling refers to more common scheduling approaches in which task dependencies are determined by functionality, risk, resources, etc. Table 1 shows the variety of combinations of size estimation, effort estimation, and scheduling approach.

By combining each of these factors with each of the five software designs, forty-two different design-scheduling scenarios were used to observe the effects of the model (in some cases, certain combinations were infeasible). Table 2 shows the statistical results of these 42 demonstrations. The data is fairly scattered. However, in all but one of the demonstration cases, a significant (i.e., 5% or greater) reduction in estimated development time was observed.

It is notable that the design refinements in each of the demonstration scenarios contributed as much or more to improving quality as to improving estimated cycle-time. The quality of each of the poor quality designs improved, while the high quality designs did not experience any loss in quality. The details of this demonstration of effectiveness can be found in [1]. It should be noted that the demonstration scenarios are all small programs, and it remains to be seen how well this model scales to large-scale software systems.

Summary and Conclusions

Current design cycles do not provide mechanisms for determining on which parts of the system to focus refinement efforts, or for determining when to stop iterating. This model provides a solution to both problems. Through a schedule analysis, the model guides the designer to focus refinement efforts on the most beneficial system components. By using the critical path as a metric of cycle-time improvement, this model helps determine when to halt the design cycle. Additionally, this model

Characteristic	Minimum	Maximum	Average	Standard Deviation
Initial Critical Path Length	12.34 days	212 days	73.98 days	54.96 days
Final Critical Path Length	10.25	122.25	51.62 days	33.94 days
Critical Path Reduction	71.91 -69.66 days	88 - 42.5 days		
Critical Path Reduction	3.13%	51.7%	29.94%	12.31%
Scheduling Iterations	1	14	5.35	2.76
Design Iterations	0	2	1.22	0.42

Table 2: Results of 42 Model Demonstrations

provides a mechanism for the designer to determine which techniques to apply in order to resolve targeted problems. In this manner, the cycle-time reduction model helps the designer *selectively apply design-improving techniques to key components in the system* in order to cost-effectively improve the development cycle time. Applying Laws of Pareto to cycle time, 20% of the software design will account for 80% of the cycle time. The goal of this model is to help software engineers focus their energy on the key 20% for improvement by identifying system components that most significantly impact the critical path length.

There are a number of other benefits to this cycle time reduction model:

- *It is not a replacement methodology.* This design approach allows software engineers to continue using state-of-the-art design and scheduling methods.
- *This method provides design-improvement feedback.* Using the critical path as a metric, the impact of design refinements is clear.
- *This method combines established practices.* Popular design heuristics, scheduling methods, and cost-estimation techniques are supported in this model.
- *This method provides motivation for iterative design refinement.* Currently, it is unclear how much benefit is gained by placing energy into design iteration. This method provides this information in the form of units of time saved.
- *The method provides a metric for measuring cycle-time reduction.* Critical path length offers a quantitative measure of the effects of the model on estimated cycle-time reduction.
- *The method allows for fixed resource allocations.* Resource constraints that limit scheduling flexibility are accommodated by this model. Furthermore, if resources are not fixed, this model provides useful information for determining resource requirements.
- *Unrealistic schedules are identified early by this model.* As the designs are refined for shorter cycle times, the feasibility of initial scheduling estimates becomes apparent allowing for contingency planning and deadline renegotiation early in the product life cycle.

Due to the small size of the software designs that were used to evaluate the model, the usefulness of the model on medium or large software designs is unclear. Further empirical evidence is required to make any conclusive statements about the model's effectiveness. However, initial results are encouraging. This design-based cycle-time reduction model can only improve as cost/effort estimation models become more accurate. Furthermore, this model is easily adaptable to organizational idiosyncrasies and to changing technologies.

Bibliography

[1] Collier, K.W. "A Design-based Model for the Reduction of Software Cycle Time", Ph.D. Dissertation, Arizona State University, 1993.

[2] Collier, K.W. and J.S. Collofello, "Issues in Software Cycle Time Reduction", *Proceedings: International Phoenix Conference on Computers and Communications*, March 28-31, 1995, pp. 302-309.

[3] Dieter, G.E. Engineering Design: A Materials and Process Approach, McGraw-Hill New York 1983.

[4] Bisant, D.B. and J.R. Lyle, "A Two-Person Inspection Method to Improve Programming Productivity", *IEEE Transactions on Software Engineering* vol. 15 no. 10, 1989, pp. 1294-1304.

[5] Boehm, B., M.H. Penedo, D.E. Stuckle, R.D. Williams and A.B Pyster, "A Software Development Environment for Improving Productivity", *Computer*, vol. 17 no. 6, 1984, pp. 30-42.

[6] Dart, S.A., R.J. Ellison and P.H. Feiler, "Software Development Environments", *Computer*, vol. 20 no. 11, 1987, pp. 18-28.

[7] Brooks, F.P. The Mythical Man-Month: Essays on Software Engineering, Addison-Wesley, Massachusetts, 1982.

[8] Boctor, F.F. "Some Efficient Multi-heuristic Procedures for Resource Constrained Project Scheduling", *European Journal of Operational Research*, vol. 49, 1990, pp. 3-13.

[9] Khattab, M.M. and F. Choobineh, "A New Approach for Project Scheduling With a Limited Resource", *International Journal of Production Research*, vol. 29 no. 1, 1991, pp. 185-198.

[10] Khattab, M.M. and F. Choobineh, "A New Heuristic for Project Scheduling With a Single Resource Constraint", *Computers and Industrial Engineering*, vol. 20 no. 3, 1991, pp. 381-387.

[11] Boehm, B. "Software Engineering Economics", *IEEE Transactions on Software Engineering* vol. SE-10 no. 1, 1984, pp. 4-21.

[12] Putnam, L.H. and W. Myers, Measures for Excellence: Reliable Software on Time, Within Budget, Yourdon Press, New Jersey, 1992.

[13] Yourdon, E. Managing the Structured Techniques, Prentice-Hall, New Jersey, 1989.