

 Supplementary files are available for this work. For more information about accessing these files, follow the link from the Table of Contents to "Reading the Supplementary Files".

# Software Process Concurrency

Raymond J. Madachy, Ph.D.

USC Center for Software Engineering  
Computer Science Department  
University of Southern California  
Los Angeles, CA 90089-0781  
213/740-5703  
madachy@usc.edu

Cost Xpert Group, Inc.  
2990 Jamacha Road, Suite 260  
San Diego, CA 92019  
619/670-6168  
madachyr@costxpert.com

## Abstract

Process concurrency provides a robust framework for modeling software processes and their constraint mechanisms. It is general enough to characterize a broad spectrum of current and emerging methodologies in terms of work available to complete on a project. It is more generally applicable than the Rayleigh curve, provides a detailed view of process dynamics and is meaningful for planning and improvement purposes. With it one can derive optimal staffing profiles for different project types, and as a shared project model it serves to improve stakeholder communication.

The software industry is continually introducing new processes, methodologies and tools. Many modern techniques serve to increase concurrency (and thus decrease cycle time) in several ways like increasing task parallelism or automating product elaboration. Process concurrency can evaluate such strategies by modeling task interdependency constraints between and within phases.

This paper will introduce process concurrency, show examples from the software development domain, compare concurrency relationships for typical development situations, run some simulation experiments, and present lessons for practitioners based on the modeling. Finally, the notions of process concurrency, Rayleigh curves and Brooks's Law are integrated from the perspective of making work available.

**Keywords:** process concurrency, software process dynamics, software process improvement, Rayleigh curve, Rapid Application Development, COTS, software engineering, Brooks's Law

## Process Concurrency Overview

Process concurrency is the degree to which work becomes available based on work already accomplished. It describes interdependency constraints between tasks, both within and between project phases. Concurrency relationships are crucial to understanding process dynamics. Internal process concurrency refers to available work constraints within a phase, while external process concurrency is used to describe available work constraints between development phases. A good treatment of process concurrency for general industry can be found in [Ford-Sterman 97], and this work interprets and extends the concepts for software engineering.

The availability of work described by process concurrency is a very important constraint on progress. Otherwise, a model driven solely by resources and productivity will allow a project to complete in almost zero time with infinite resources. Such is not the case with software processes where tasks are highly interdependent, since some tasks must be sequential and can't be done in parallel.

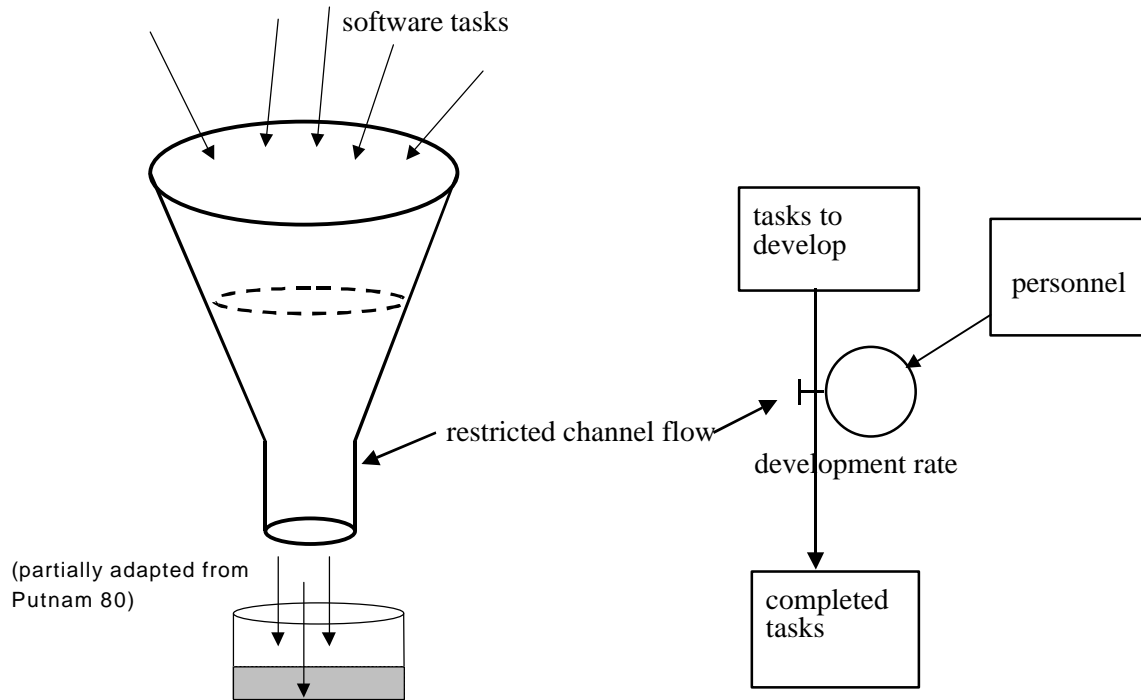
Process concurrence relationships describe how much work becomes available for completion based on previous work accomplished. These realistic bottlenecks on work availability should be considered during project planning and execution. There is a limit to the amount of concurrent development due to interdependencies in software processes. Concurrence relations can be sequential, parallel, partially concurrent, or other dependent relationships. Concurrence relationships can be elicited from process participants. A protocol for the elicitation is described in [Ford-Sterman 98].

The definition of "task" in this context is an atomic unit of work that flows through a project, where the units may differ among project phases. This is the same treatment of tasks used in the Abdel-Hamid model and many others. Tasks are assumed to be interchangeable and uniform in size (e.g. the Abdel-Hamid task was equivalent to 60 lines of software). A task then refers to product specification during project definition, and lines of code during code implementation. The assumption becomes more valid as the size of task decreases.

### ***Trying to Accelerate Software Development***

It is instructive to understand some of the phenomena that impede software processes. Putnam likens the acceleration of software development to pouring water into a channel-restricted funnel [Putnam 80]. The funnel does not allow the flow to be sped up very much, no matter how much one pours into the funnel. This is like throwing a lot of software personnel at once into the development chute to accelerate things. They won't be able to work independently in parallel, since certain tasks can only be done in sequence.

Figure 1 shows the limited parallelism of software tasks using a funnel analogy alongside the corresponding system dynamics structure. This concept is elaborated in the next Figure 2, which shows the constriction brought about by trying to parallelize sequential (or at least partially sequential) activities for a single thread of software. Tasks can only flow through in proper order. This image is reminiscent of the *Three Stooges* getting stuck while trying to enter a single doorway at the same time.

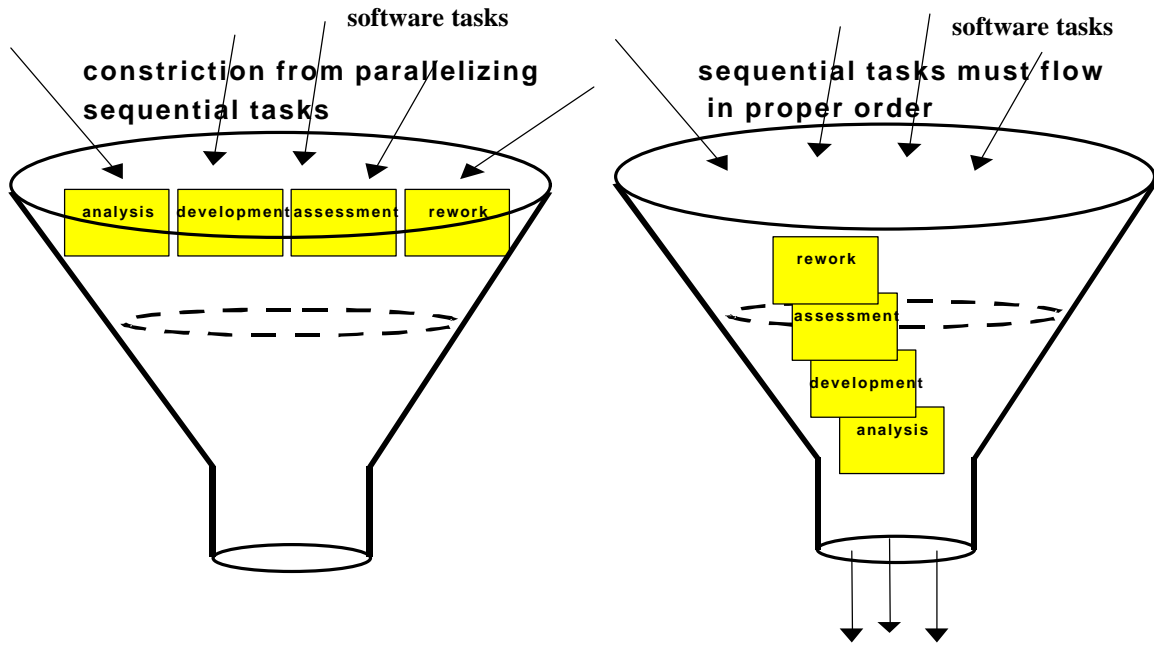


**Figure 1: Funnel View of Limited Task Parallelism and System Dynamics Corollary**

There are always sequential constraints independent of phase. The elemental activities in any phase of software development include:

- analysis and specification; one figures out what you're supposed to do and specifying how the parts fit together
- development of something (architecture, design, code, test plan, etc.) that implements the specifications
- assessment of what was developed; this may include verification, validation, review, or debugging
- possible rework recycle of previous activities

These activities can't be done totally in parallel with more applied people. Different people can perform the different activities with limited parallelism, but downstream activities will always have to follow some of the upstream.



**Figure 2: Trying to Parallelize Sequential Tasks in the Funnel**

In *The Mythical Man-Month* [Brooks 95], Brooks explains these restrictions from a partitioning perspective in his Brooks's Law framework. Sequential constraints imply tasks cannot be partitioned among different personnel resources. Thus applying more people has no effect on schedule. Men and months are interchangeable only when tasks can be partitioned with no communication among them. Process concurrence is a natural vehicle for modeling these software process constraints.

### ***Internal Process Concurrence***

An internal process concurrence relationship shows how much work can be done based on the percent of work already done. The relationships represent the degree of sequentiality or concurrence of the tasks aggregated within a phase. They may include changes in the degree of concurrence as work progresses. Figure 3 and Figure 4 demonstrate linear and non-linear internal process concurrence. The bottom right half under the diagonal of the internal process concurrence is an infeasible region, since the percent available to complete can't be less than the amount already completed.

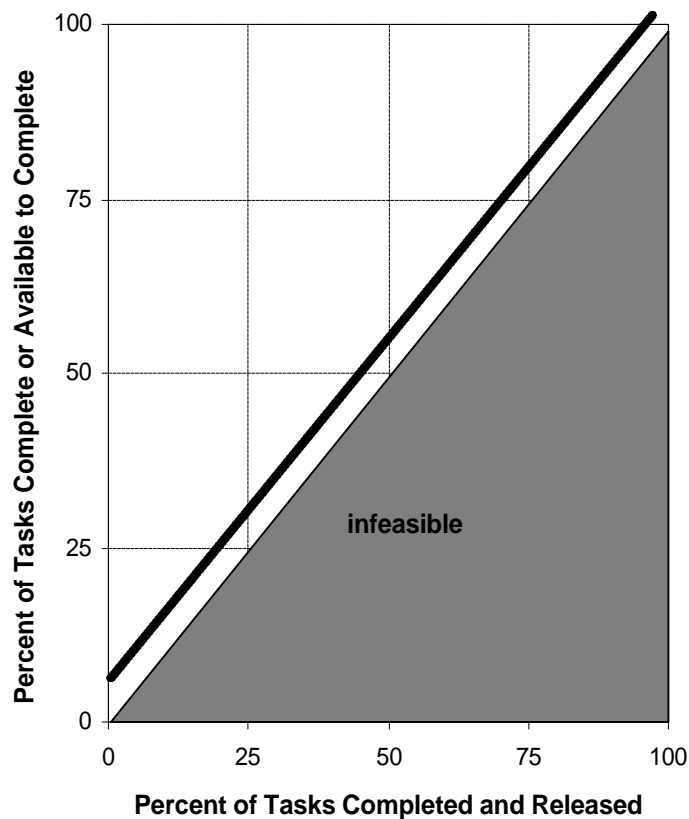
The development of a single software task within a phase normally includes the sequences of construction, verification/validation and sometimes rework. These can't all be simultaneously performed on an individual task. There will always be an average duration of the sub-activities, regardless of the resources applied to them. Additionally, the development of some tasks require the completion of other intra-phase tasks beforehand. Thus, the process limits the availability of work to be completed based on the amount of already completed work.

More concurrent processes are described by curves near the left axis, and less concurrent processes lie near the 45° line. The linear relationship in Figure 3 could describe the sequential construction of a 10-story building. When the first floor is complete, 10% of the project is done and the second floor is available to be completed, or 20% of the entire project is thus available to

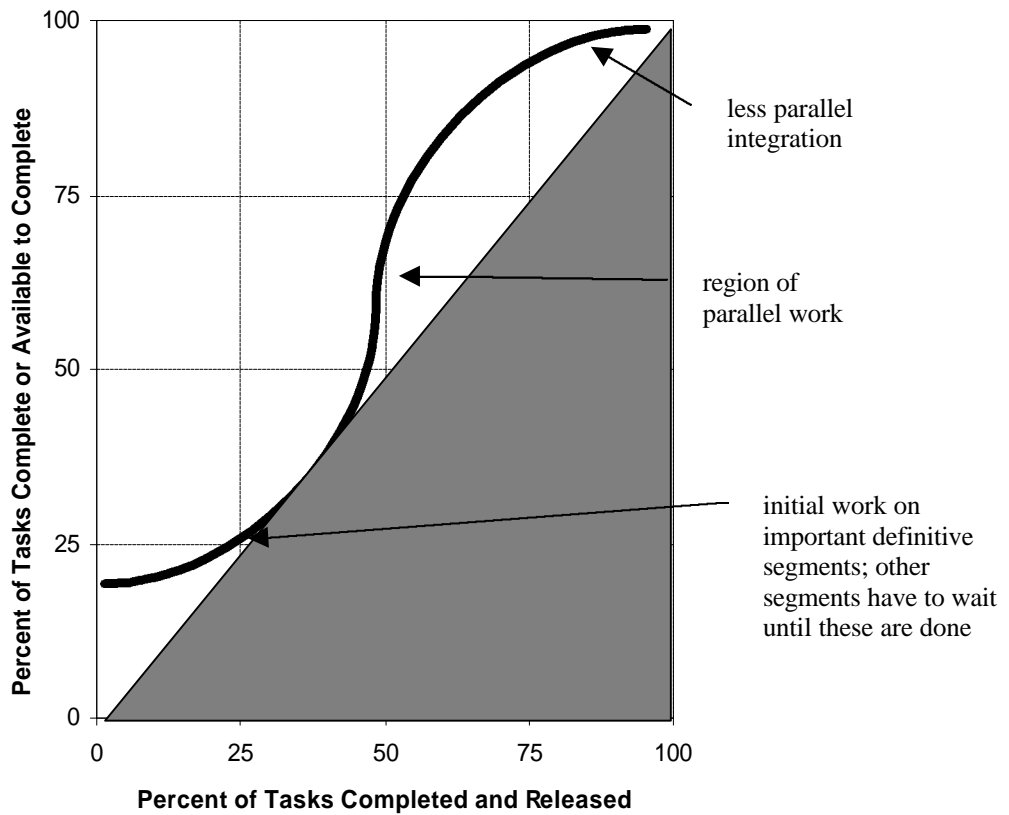
finish. This pattern continues until all the floors are done. This is sometimes called a “lockstep” relationship.

The linear concurrence line starts above the 45 degree diagonal, since the y-axis includes work "available to complete". The relationship has to start greater than zero. Consider again a skyscraper being built. At the very beginning, the first floor is available to be completed.

A more typical non-linear relationship is shown in Figure 4. For example, the overarching segments of software must be completed before other parts can begin. Only the important portions (such as 20% of the whole for an architecture skeleton, interface definitions, common data, etc.) can be worked on in the beginning. The other parts aren't available for completion until afterwards. This typifies complex software development tasks where many tasks are dependent on each other. Men and months are not interchangeable in this situation according to Brooks because the tasks cannot be partitioned without communication between them.

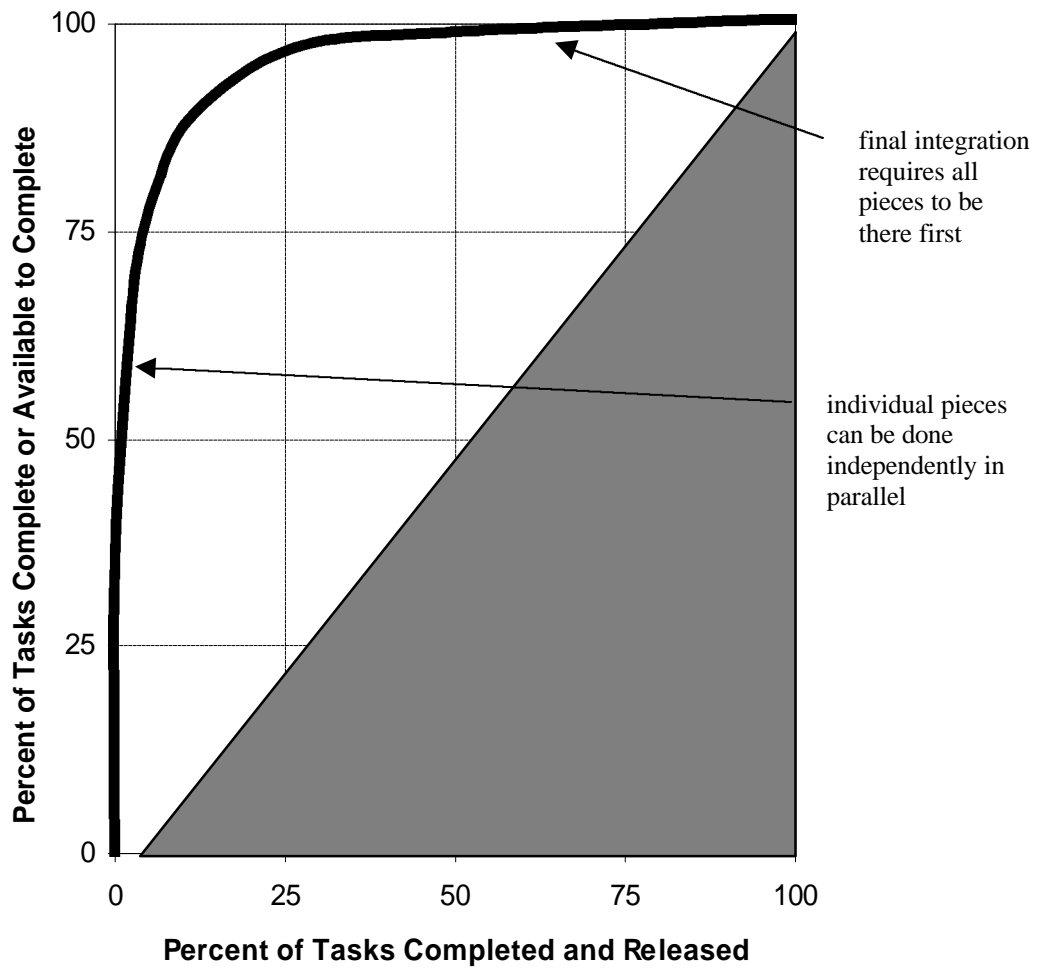


**Figure 3: Linear Internal Process Concurrence**



**Figure 4: Non-linear Internal Process Concurrence**

Figure 5 shows internal concurrence for an extreme case of parallel work. There is very high concurrency because the tasks are independent of each other. Almost everything can be doled out as separate tasks in the beginning, such as a straightforward translation of an existing application from one language to another. Each person simply gets an individual portion of code to convert, and that work can be done in parallel. The last few percent of tasks for integrating all translated components have to wait until the different pieces are there first, so 100% can't be completed until the translated pieces have been completed and released. Brooks explains that many tasks in this situation can be partitioned with no communication between them, and men and months are largely interchangeable.



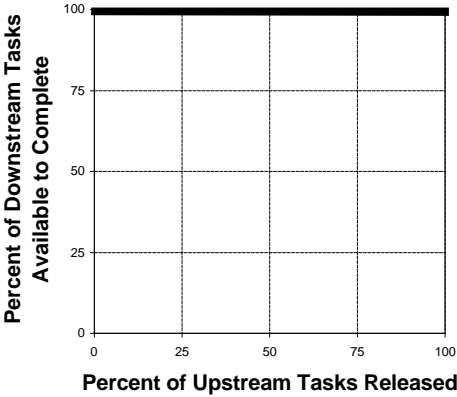
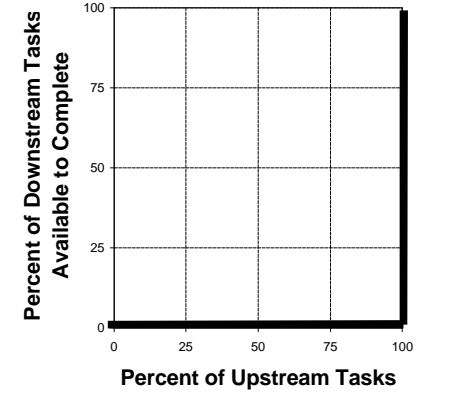

**Figure 5: Nearly Parallel Internal Process Concurrence**

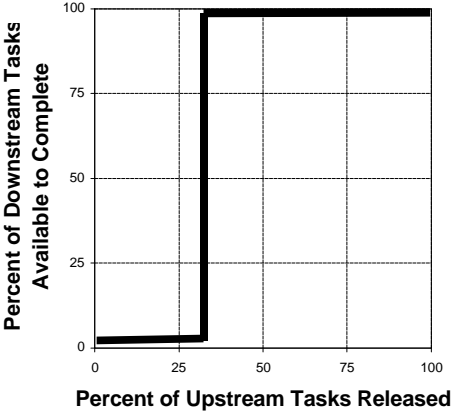
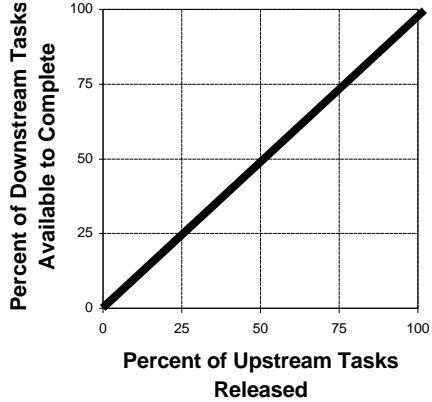
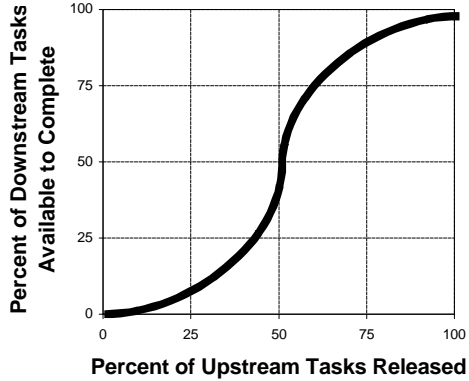
***External Process Concurrence***

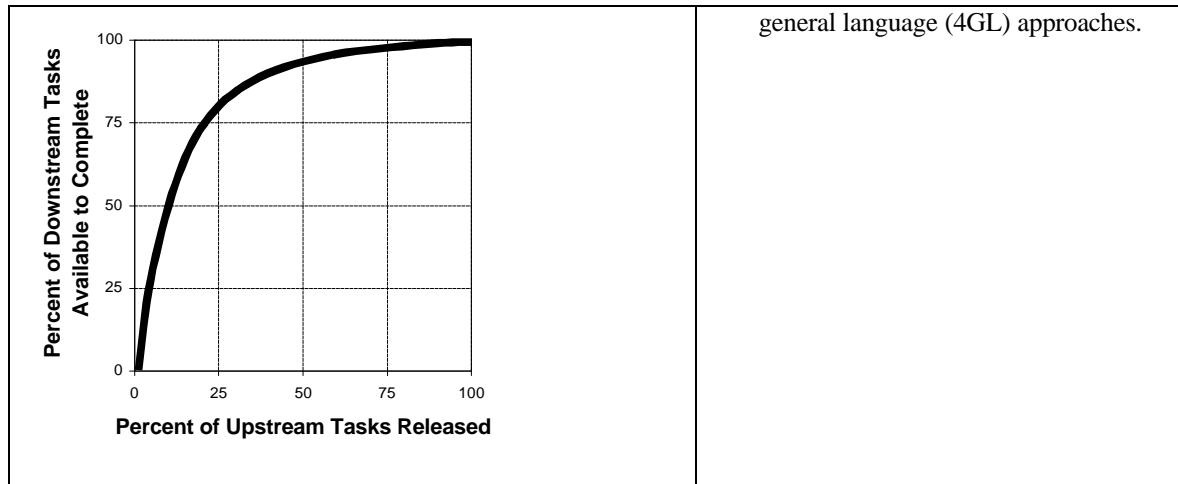
External process concurrence relationships describe constraints on amount of work that can be done in a downstream phase based on the percent of work released by an upstream phase. Examples of several external process concurrence relationships are shown in Table 1. More concurrent processes have curves near the upper left axes, and less concurrent processes have curves near the lower and right axes.



**Table 1: External Process Concurrency Relationships**

Relationship	Characteristics
<p>No Inter-phase Relationship</p> 	<ul style="list-style-type: none"> <li>• No dependencies between the phases.</li> <li>• The downstream phase can progress independently of the upstream phase.</li> <li>• The entire downstream work is available to be completed with none of the upstream work released.</li> </ul>
<p>Sequential Inter-phase Relationship</p> 	<ul style="list-style-type: none"> <li>• None of the downstream phase can occur until the upstream phase is totally complete.</li> <li>• Like a theoretical waterfall development process where no phase can start until the previous phase is completed and verified.</li> <li>• Same as a finish-stop relationship in a critical path network.</li> </ul>
<p>Parallel Inter-phase Relationship</p> 	<ul style="list-style-type: none"> <li>• The two phases can be implemented completely in parallel.</li> <li>• The downstream phase can be completed as soon as the upstream phase is started.</li> </ul>
<p>Delayed Start Inter-phase Relationship</p>	<ul style="list-style-type: none"> <li>• The downstream phase must wait until a major portion of the upstream phase is completed, then it can be completed in its entirety.</li> </ul>

 <p>Percent of Downstream Tasks Available to Complete</p> <p>Percent of Upstream Tasks Released</p>	<ul style="list-style-type: none"> <li>• Like a start-start relationship in a critical path network.</li> </ul>
<p>Lockstep Relationship</p>  <p>Percent of Downstream Tasks Available to Complete</p> <p>Percent of Upstream Tasks Released</p>	<ul style="list-style-type: none"> <li>• The downstream phase can progress at the same speed as the upstream phase; thus they are in lockstep with each other.</li> <li>• The downstream work availability is correlated linearly 1:1 to how much is released from the upstream. For example, after 10% of system design is completed then 10% of implementation tasks is available to finish.</li> <li>• This relationship is not available in PERT/CPM.</li> </ul>
<p>Delay with Partially Concurrent Inter-phase Relationship</p>  <p>Percent of Downstream Tasks Available to Complete</p> <p>Percent of Upstream Tasks Released</p>	<ul style="list-style-type: none"> <li>• The downstream phase has to wait until a certain percentage of upstream tasks have been released, and then can proceed at varying degrees of concurrence per the graph.</li> <li>• This relationship is representative of complex software development with task interdependencies.</li> <li>• This type of relationship is not available with PERT/CPM methods</li> </ul>
<p>Leveraged Concurrence Relationship</p>	<ul style="list-style-type: none"> <li>• This relationship exhibits a high degree of parallelism and leverage between phases.</li> <li>• Typical of Commercial Off-The Shelf (COTS) products whereby one specifies the capabilities, and the system is quickly configured and instantiated. Also applicable for 4<sup>th</sup></li> </ul>



The partially concurrent inter-phase relationship is representative of much software development, where complexities impose task dependencies and thus inter-task communication is necessary. For example, a critical mass of core requirements must be released before architecture and design can start. Then the downstream phase availability rate increases in the middle region (e.g. much design work can take place), and then slows down as the final upstream tasks are released.

External process concurrence relationships function like the precedence relationships in critical path and PERT methods to describe dependencies, but contain greater dynamic detail. For example, external concurrence relationships describe the phase dependencies for the entire durations. PERT and critical path methods only use the stop and start dates. They can also be nonlinear to show differences in the degree of concurrence, whereas PERT methods cannot. Lastly, process concurrence relationships are dynamic since the work completed could increase or reduce over time, but only static precedence relationships are used in critical path or PERT methods. The lockstep and delay with partially concurrent inter-phase relationships are situations that cannot be described with PERT or critical path methods.

### ***Analysis of Different Software Methodologies***

We'll examine different methodologies via process concurrence in terms of leverage between phases. Leverage in software development refers to how much can be elaborated based on inputs from the previous phase. Software development is essentially a transformational process whereby the product goes from concept to requirements to design and finally code. A method with increased leverage means that the downstream product can be elaborated with less effort. It is like a return-on-investment for a given effort, or can be likened to mechanical advantage.

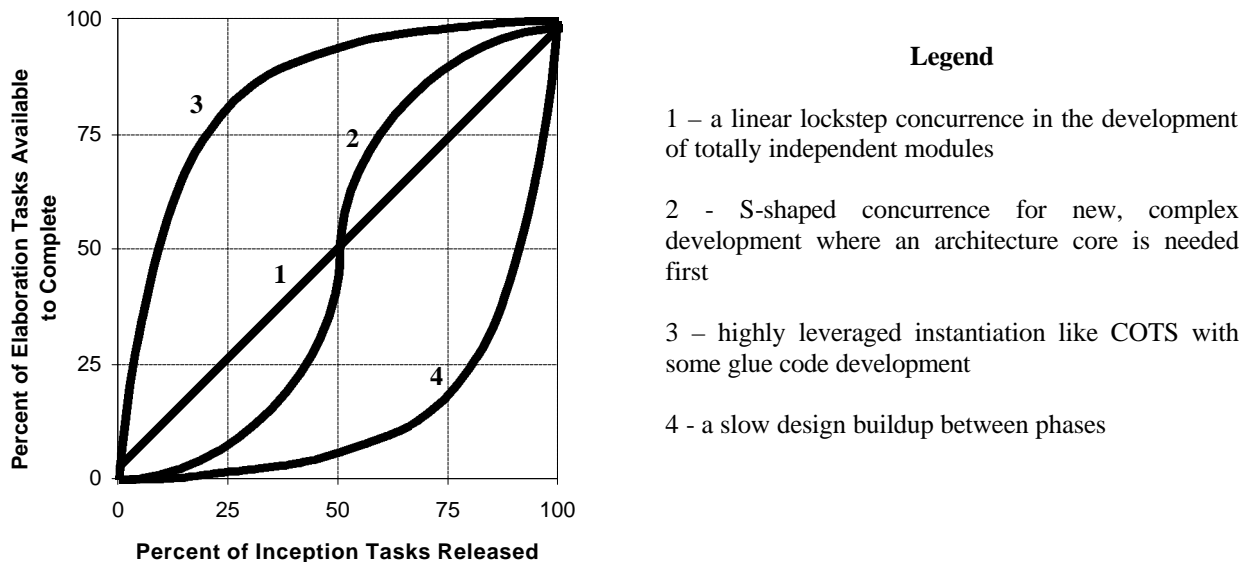
For example, a Fourth-Generation Language (4GL) that generates code from requirement statements has much higher leverage than new code development. For about the same amount of effort expended on requirements, a great deal more of demonstrative software will be produced with advanced 4GL tools compared to starting from scratch in each phase. Starting from scratch means there are no pre-existing software artifacts, and all development is new. Virtually all modern approaches to software development are some attempt to increase leverage, so that

machines or humans can efficiently instantiate software artifacts vs. more labor-intensive approaches.

Commercial-Off-The-Shelf (COTS) software is another good example of high phase leverage, because functionality is easily created after identifying the COTS package. If one specifies “use the existing SIRSI package for library operations”, then the functions of online searching, checkout, etc. are already defined and easily implemented after configuring the package locally.

Process concurrence is a very useful means to contrast the leverage of different approaches, because the degree of concurrence is a function of the software methodology (among other things). When developing or interpreting external process concurrence curves for software development strategies, it is helpful to think of tasks in terms of demonstrable functionality. Thus tasks released or available to complete can be considered analogous to function points in their phase-native form (e.g. design or code). With this in mind, it is easier to compare different strategies in terms of leverage in bringing functionality to bear.

Typical examples are shown in Figure 6 to visualize the contrasts between some different situations. More details are provided in the following examples explaining the curves.



**Figure 6: Examples of External Concurrence**

### *RAD Example of External Process Concurrence*

Increasing task parallelism is a primary opportunity to decrease cycle time in RAD. Process concurrence is ideally suited for evaluating RAD strategies in terms of work parallelism constraints between and within phases. System dynamics is very attractive to analyze schedule time in this context vs. other methods, because it can model task interdependencies on the critical path. Only those tasks on the critical path have influence on the overall schedule.

One way to achieve RAD is by having base software architectures tuned to application domains available for instantiation, standard database connectors and reuse. This example demonstrates how the strategy of having pre-defined and configurable architectures for a

problem domain can increase the chance for concurrent development between inception and elaboration.

*Developing from Scratch*

Suppose the software job is to develop a Human Resources (HR) self-service portal. It is web-based system for employees in a large organization to access and update all their personnel and benefits information. It will have to tie into existing legacy databases and commercial software packages for different portions of the human resources records. The final system will consist of the following:

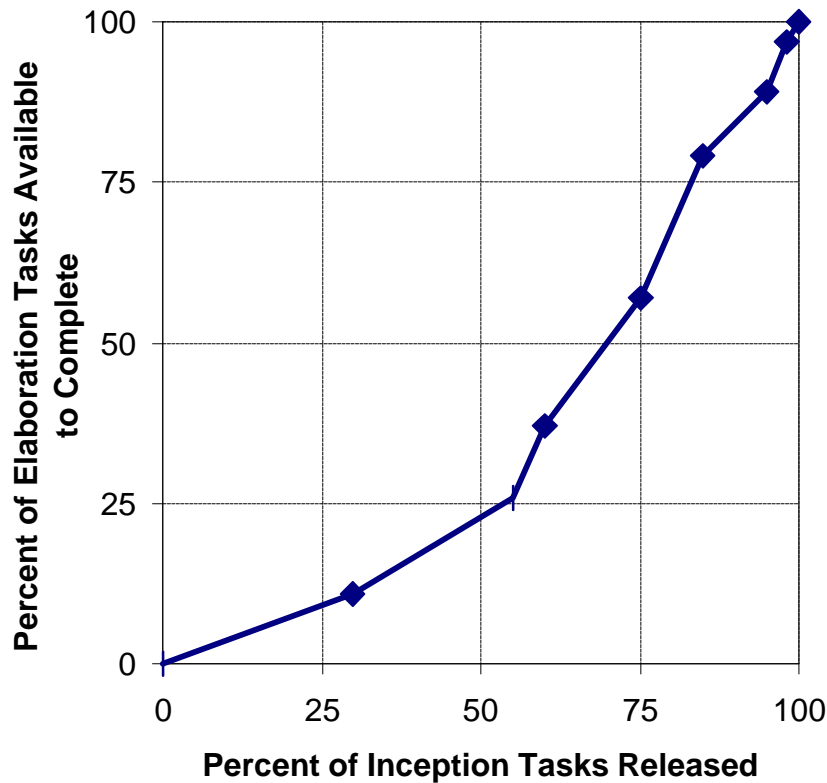
- 30% user interface (UI) front-end
- 30% architecture and core processing logic
- 40% database (DB) wrappers and vendor package interface logic (back-end processing).

Table 2 describes an example concurrence relationship between inception and elaboration for this system, where inception is defining the system capabilities and elaboration is designing it for eventual construction. The overall percent of tasks ready to elaborate is a weighted average. There is no base architecture from which to start from.

**Table 2: Concurrence Worksheet for Developing HR Portal from Scratch**

Inception (System Definition)		Elaboration (System Design)	
Requirements Released	% of Inception Tasks Released	% of Components Ready to Elaborate	Overall % of Tasks Ready to Elaborate
About 25% of the core functionality for the self-service interface supported by prototype. Only general database interface goals defined.	30%	20% UI 10% core 5% DB	11%
About half of the basic functionality for the self-service interface supported by prototype.	55%	40% UI 20% core 20% DB	26%
Interface specifications to Peoplesoft defined for internal personnel information.	60%	40% UI 30% core 40% DB	37%
More functionality for benefits capabilities defined (80% of total front-end)	75%	75% UI 60% core 40% DB	57%
Interface specification to JD Edwards and SAP systems for life insurance and retirement information.	85%	75% UI 80% core 80% DB	79%
Rest of user interface defined (95% of total), except for final UI refinements after more prototype testing.	95%	95% UI 95% core 80% DB	89%
Timecard interface to Oracle system defined.	98%	95% UI 95% core 100% DB	97%
Last of UI refinements released.	100%	100% UI 100% core 100% DB	100%

Figure 8 shows a plot of the resulting external concurrence relationship from this worksheet.

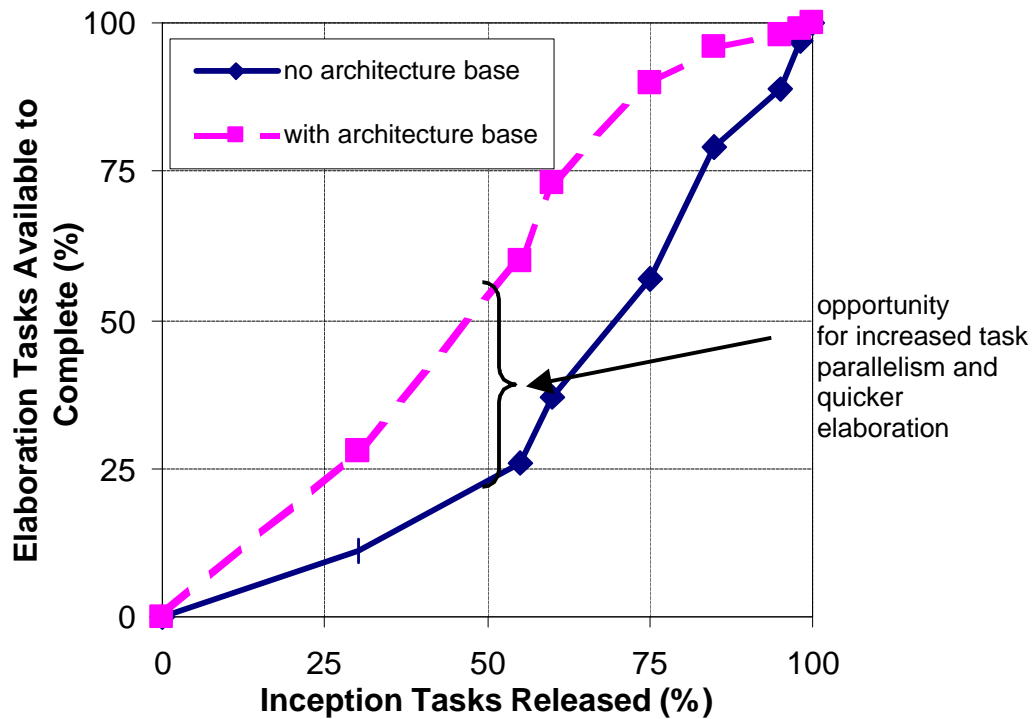


**Figure 7: External Process Concurrence for HR System from Scratch**

### *Developing With a Base Architecture*

Contrast the previous example with another situation whereby there exists a base architecture that has already been tuned for the Human Resources application domain of processes and employee service workflows. It uses XML technology that ties to existing database formats and is ready for early elaboration by configuring the architecture. It already has standard connectors for different vendor database packages.

Figure 9 shows the corresponding process concurrence against the first example where an architecture had yet to be developed. This relationship enables more parallelism between the inception and elaboration phases, and thus the possibility of reduced cycle time. When about 60% of inception tasks are released, this approach allows 50% more elaboration tasks to be completed vs. the development from scratch.



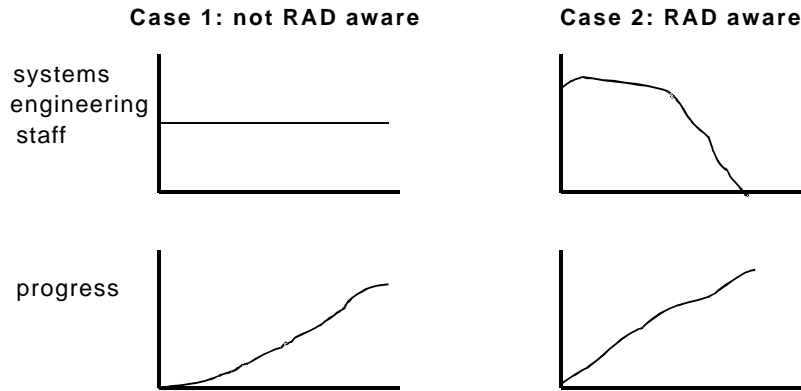
**Figure 8: External Process Concurrency Comparison with Base Architecture**

### *RAD Systems Engineering Staffing Considerations*

Staffing profiles are often dictated by the staff available, but careful tuning of the people dedicated to particular activities can have a major impact to overall schedule time. Consider the problem early in a project when developers are “spinning their wheels” and wasting time while the requirements are being derived for them. Typically the requirements come from someone with a systems engineering focus. If those people aren’t producing at a rate fast enough for the software people to start elaborating on, then effort is wasted. The staffing plan should account for this early lack of elaboration, so that implementers are phased in at just the right times when tasks become available to complete.

Knowledge of process concurrency for a given project can be used to carefully optimize the project. The right high-level analysts (normally a small number) have to be in place producing specifications before hordes of programmers come in to implement the specifications. This optimizing choreography requires fine coordination to have the right people at the right time.

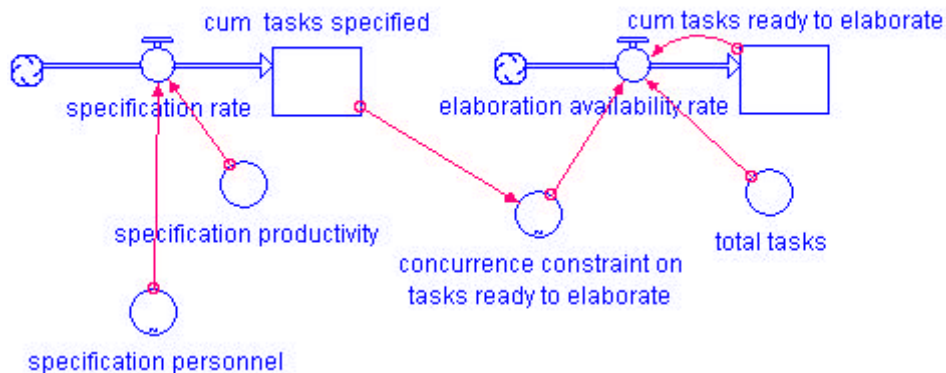
To optimize schedule on a complex project with partial inter-phase concurrency, the optimal systems engineering staffing is front-loaded vs. constant level-of-effort. As shown by process concurrency, the downstream development is constrained by the specifications available. Figure 10 shows two situations of RAD awareness. In the first case, there is a non-optimal constant staff level for systems engineering and overall progress is impeded. If a curvilinear shape is used instead to match the software development resources, then cycle-time gains are possible because programming can complete faster.



**Figure 9: Illustration of RAD Awareness to Systems Engineering Staffing**

### *External Concurrence Model and Experimentation*

A simple model of external process concurrence is shown in Figure 11 representing task specification and elaboration. It models concurrence dynamics in the elaboration phase of a typical project, and will be used to experimentally derive staffing profiles for different combinations of specification inputs and concurrence types. The specification personnel is a forcing function to the model, and is simply a graphical profile that can take on various staffing shapes. The specification input profile mimics how requirements are brought dynamically into a project. The concurrence constraint is also a graphical function drawn to represent different process concurrence relationships to be studied.



**Figure 10: External Concurrence Model**

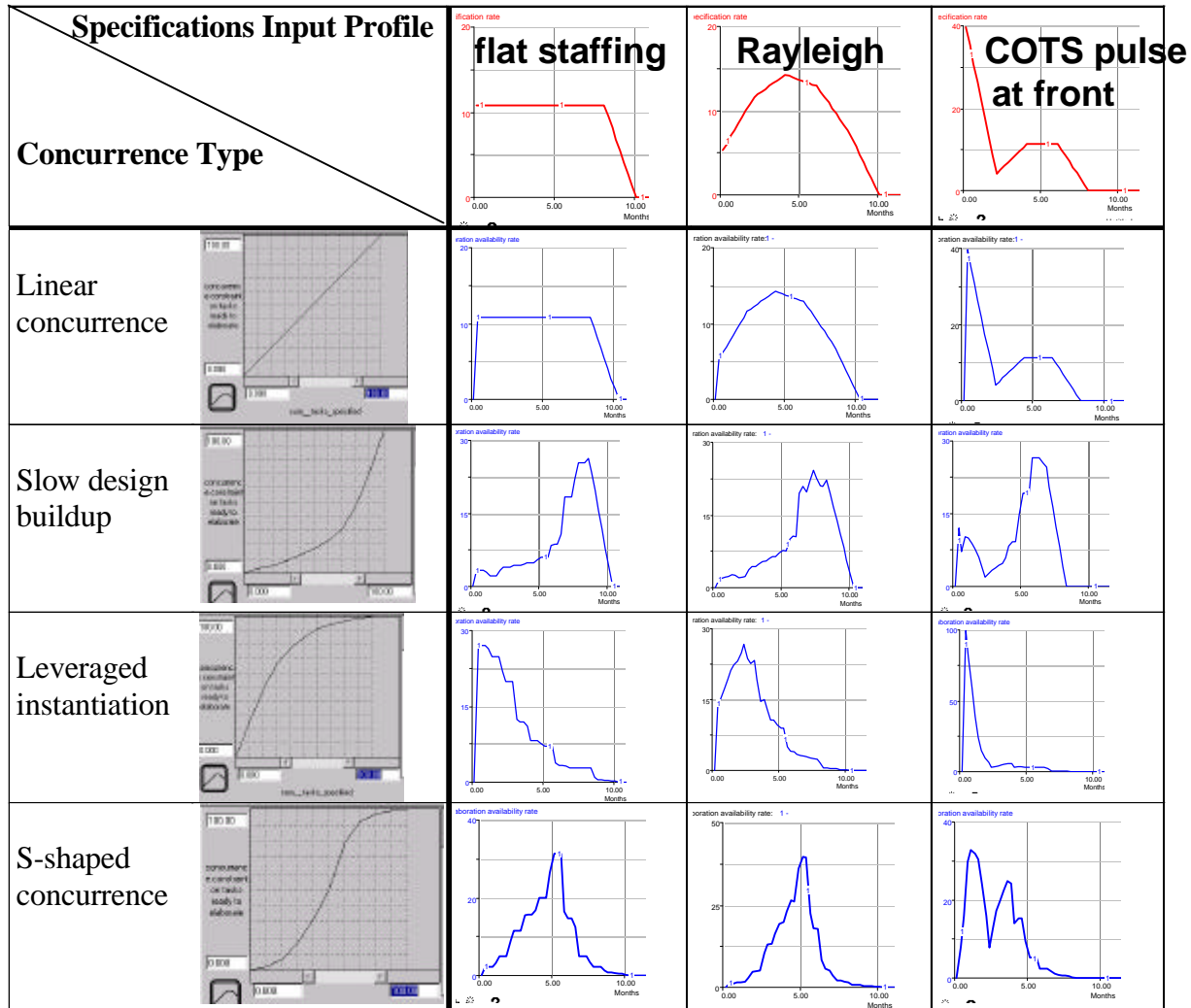
The time profile of tasks ready to elaborate will be considered proportional to an “ideal” staffing curve. This follows from an important assumption used in the Rayleigh curve that that an optimal staffing is proportional to the number of problems ready for solution. It is very important to note that this only considers the product view. The real-world process of finding and bringing people on board may not be able to keep up with the hypothetical optimal curve.



Thus the personnel perspective may trump the product one. Much experience in the field points out that a highly peaked Rayleigh curve is often too aggressive to staff to.

The model is used to experimentally derive optimal elaboration staffing profiles (from a product perspective) for different types of projects. We will explore various combinations of specification profiles and concurrence between specification and elaboration to see their effect. The staffing inputs include 1) flat staffing, 2) a peaked Rayleigh-like staffing and 3) a COTS requirements pulse at the beginning followed by a smaller Rayleigh curve to mimic a combined COTS and new development. In addition, we will vary the concurrence types as such: 1) linear lockstep concurrence, 2) a slow design buildup between phases, 3) leveraged instantiation to model COTS, and 4) S-shaped concurrence that models a wide swath of development. COTS is a pulse-like input because the requirements are nearly instantly defined by the existing capabilities.

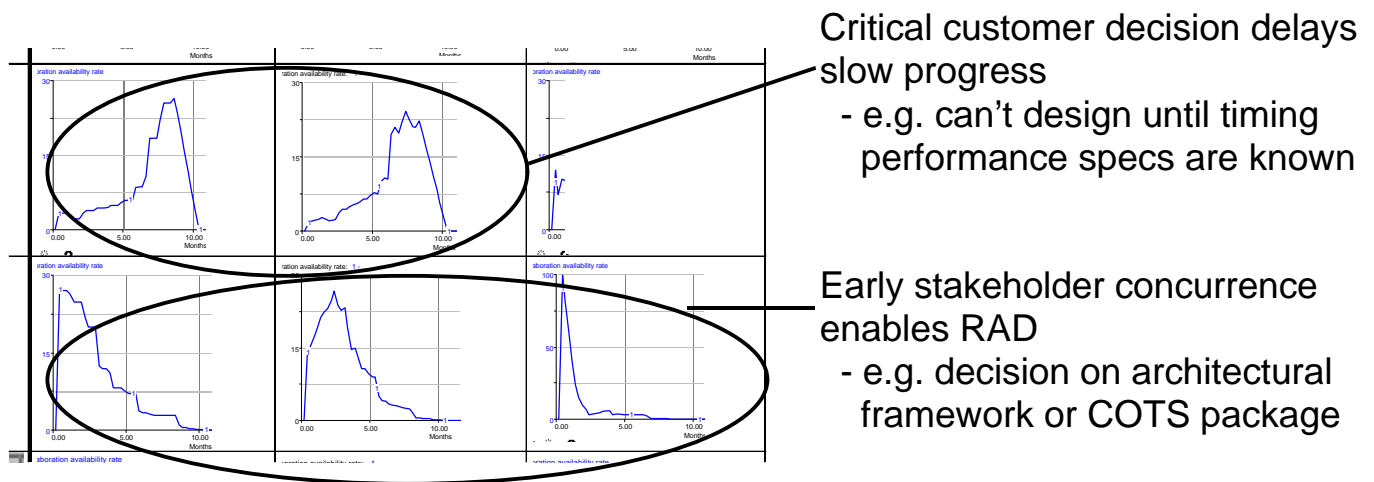
Figure 12 shows outputs of the external concurrence model for a variety of situations. It is clear that optimal staffing profiles can be far different than a standard Rayleigh curve, though some of the combinations describe projects that can be modeled with a Rayleigh curve.



**Figure 11: Elaboration Availability Simulation Results**

*Sample Lessons*

This experiment points out some lessons for practitioners. Figure 13 shows some of the simulation results with appropriate lessons. With a slow design buildup, it is clear that critical design delays will slow progress down the road since little is ready for elaboration. Alternately, rapid development can occur in a leveraged instantiation situation with early stakeholder concurrence. These both indicate that earlier architectural decisions and stakeholder agreements are important to shorten schedule. Also note that “not applicable” is shown on the output for a slow design buildup with COTS, because it violates the principles of COTS that the system is very quickly elaborated.



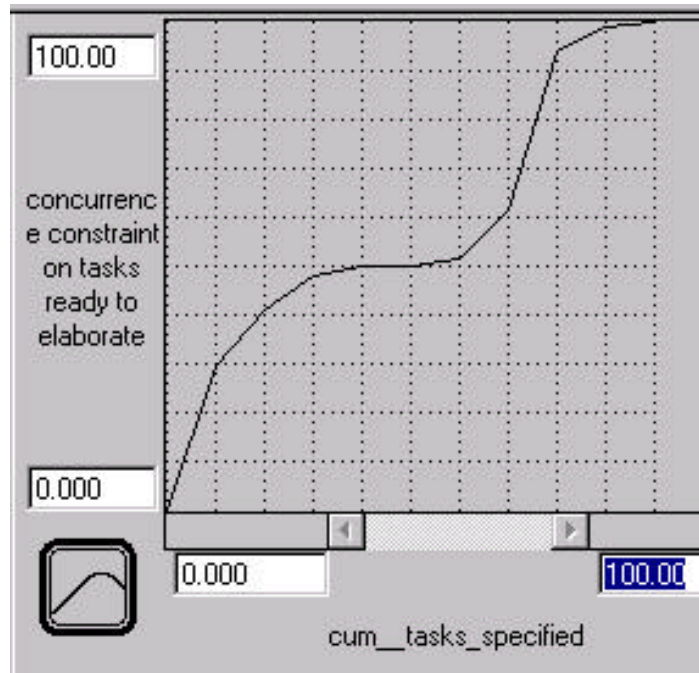
**Figure 12: Some Lessons Learned**

These simulation outputs don't tell the entire story about staffing needs because some product parts are easier to develop than others, or even automatable. To further use these results for planning a real project, the ideal staffing curves have to be modulated by relative effort. That is, implementing COTS or reused components generally require much less effort than new development. If a reused component takes 20% of the effort compared to new, then the required staffing should be similarly reduced. So the staffing curves need further adjustments for the relative effort of different approaches.

**Additional Considerations**

Process concurrence curves can be more precisely matched to the software system types. For example, COTS by definition should exhibit very high concurrence but an overall system can be a blend of approaches. Mixed strategies produce combined concurrence relationships. Concurrence for COTS first then new development would be similar to that seen in Figure 14. The curve starts out with a leveraged instantiation shape, then transitions to a slow design buildup. This type of concurrence would be better matched to the system developed with an

initial COTS pulse followed by new software development, corresponding to the third column in Figure 12. Many permutations of concurrence are possible for realistic situations.



**Figure 13: Sample Process Concurrence for Mixed Strategies**

## Rayleigh Manpower Distribution

The Rayleigh curve (also called a Norden/Rayleigh curve) is a popular model of personnel loading that naturally lends itself to system dynamics modeling. It serves as a simple generator of time-based staffing curves that is easily parameterized to enable a variety of shapes. The Rayleigh curve is actually a special case of the Weibull distribution, and serves as a model for a number of phenomena in physics.

After analyzing hardware research and development projects, Norden put forth a manpower model based on Rayleigh curves. According to these staffing curves, only a small number of people are needed at the beginning of a project to carry out planning and specification. As the project progresses and more detailed work is required, the number of staff builds up to a peak. After implementation and testing, the number of staff required starts to fall until the product is delivered. Putnam subsequently applied the Rayleigh curve to software development [Putnam 80], and it is used in several software cost models.

One of the underlying assumptions is that the number of people working on a project is approximately proportional to the number of problems ready for solution at that time. Norden derived a Rayleigh curve that describes the rate of change of manpower effort per the following first order differential equation:

$$\frac{dC(t)}{dt} = p(t)[K - C(t)]$$

where  $C(t)$  is the cumulative effort at time  $t$ ,  $K$  is the total effort, and  $p(t)$  is a product learning function. The time derivative of  $C(t)$  is the manpower rate of change, which represents the number of people involved in development at any time. Operationally it is the staff size per time increment (traditionally the monthly headcount or staffing profile). The learning function is linear and can be represented by

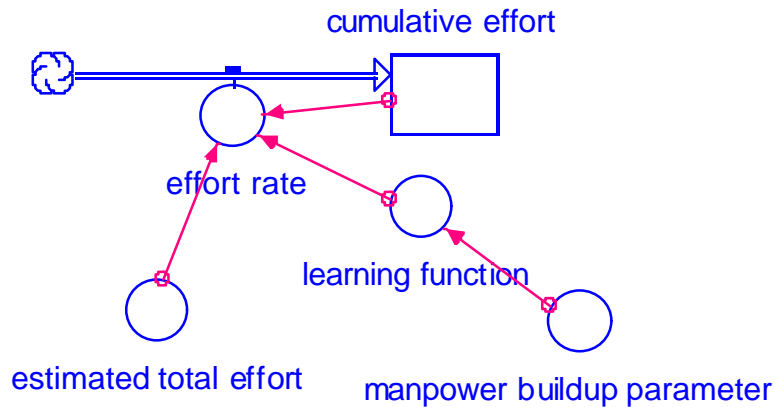
$$p(t) = 2at$$

where  $a$  is a positive number. The  $a$  parameter is an important determinant of the peak personnel loading called the manpower buildup parameter. The second term  $[K - C(t)]$  represents the current work gap; it is the difference between the final and current effort that closes over time as work is accomplished.

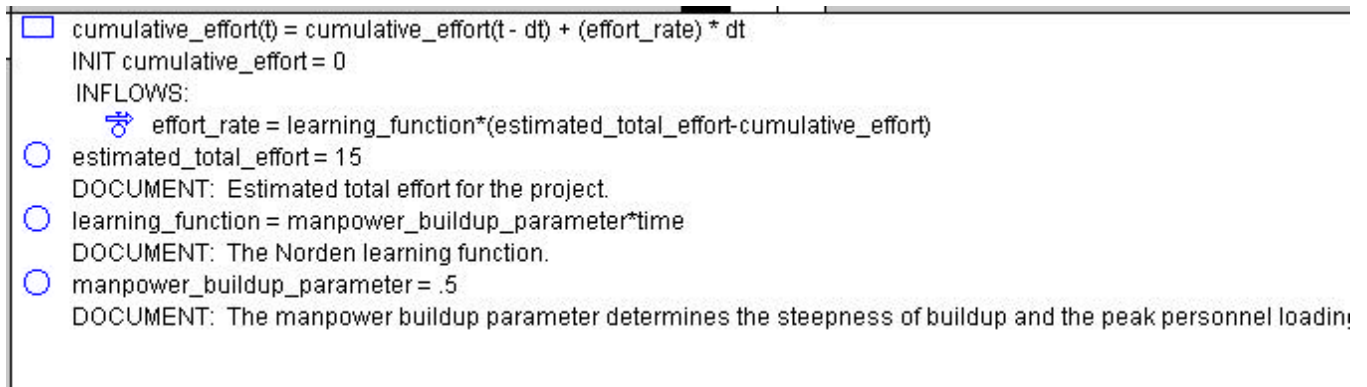
## System Dynamics Implementation

The Rayleigh curve and much of Putnam's work is naturally suited to dynamic modeling. Quantities are expressed as first and second order differential equations; precisely the rate language of system dynamics. Figure 14 and Figure 15 show a very simple model of the Rayleigh curve using an effort flow chain. The formula for manpower rate of change

represents the project staffing profile. It uses feedback from the cumulative effort (this feedback represents knowledge of completed work).



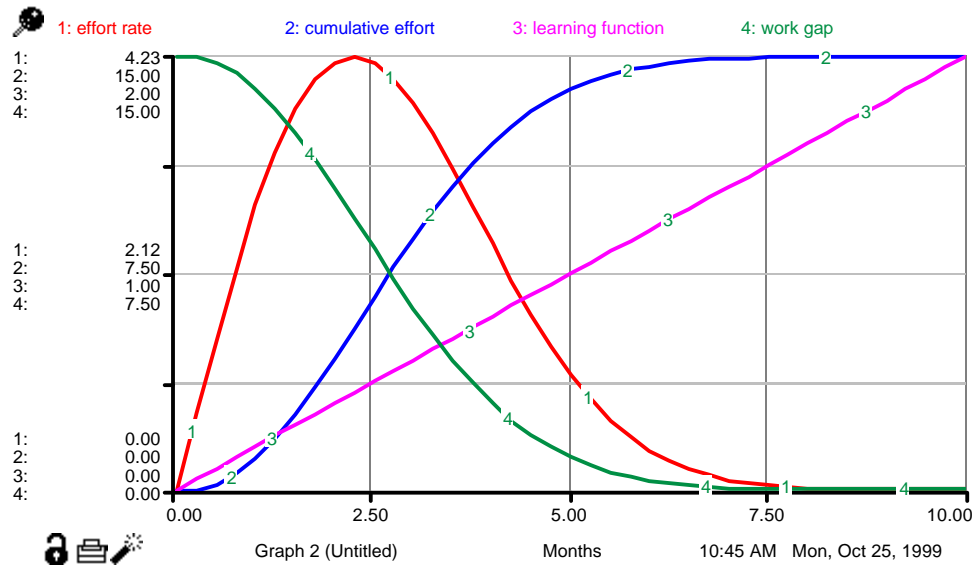
**Figure 14: Rayleigh Manpower Model**



**Figure 15: Rayleigh Manpower Model Equations**

Figure 16 shows the components of the Rayleigh formula from the simulation model. The learning function increases monotonically while the work gap diminishes over time as problems are worked out, and the corresponding effort rate rises and falls in a Rayleigh shape. The two multiplicative terms, the learning function and the work gap, produce the Rayleigh effort curve when multiplied together. They offset each other because the learning function rises and the work gap falls over time as a project progresses.

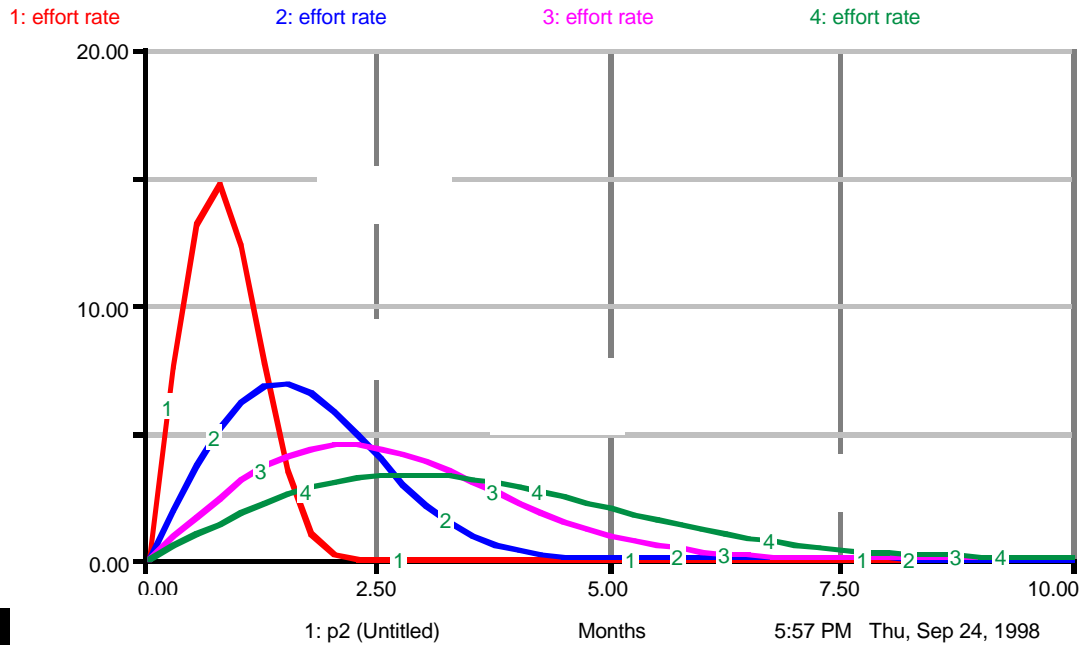
The Rayleigh is an excellent example of a structure producing S-curve behavior for cumulative effort. The learning function and work gap combine to cause the Rayleigh effort curve, which integrated over time as cumulative effort is sigmoidal. The cumulative effort starts with a small slope (the learning function is near zero), increases in slope, and then levels out as the work gap starts nearing zero.



**Figure 16: Rayleigh Curve Components**

The learning function is linear, and really represents the continued elaboration of product detail (e.g. specification to design to code), or increasing understanding about the product makeup. A true learning curve has a different non-linear shape. We prefer to call the term an “elaboration function” to better represent the true phenomena being modeled. This is also consistent with the assumption that the staff size is proportional to the number of problems (or amount of detail) ready to be implemented. This is the difference between what has been specified (the elaboration function) and what is left to do (the work gap). Other than this section that introduces traditional terminology for Rayleigh curves, we will use the term elaboration function in place of learning function.

Figure 17 shows the output staffing profile for different values of  $a$ . It is seen that the manpower buildup parameter  $a$  is an important determinant of the peak personnel loading. The larger the value of  $a$ , the earlier the peak time and a corresponding steeper profile. It is also called the manpower buildup parameter (MBP). A large  $a$  denotes a responsive, nimble organization. The qualitative effects of the MBP are shown in Table 3.



**Figure 17: Rayleigh Manpower Model Output for Varying Learning Functions**

**Table 3: Effects of Manpower Buildup Parameter**

Manpower Buildup Parameter	Effort Effect	Schedule Effect	Defect Effect
Low (slow staff buildup)	Lower	Higher	Lower
Medium (moderate staff buildup)	↓	↑	↓
High (aggressive staff buildup)	Higher	Lower	Higher

It has been observed that the staffing buildup rate is largely invariant within a particular organization due to a variety of factors. Some organizations are much more responsive and nimble to changes than others. Design instability is the primary cause for a slow buildup. Top architects and senior designers must be available to produce a stable design. Hiring delays and the inability to release top people from other projects will constrain the buildup to a large extent. If there is concurrent hardware design in a system, instability and volatility of the hardware will limit the software design ready for programming solutions.

The Rayleigh curve can be calibrated to static cost models and used to derive dynamic staffing profiles. See the model file *Rayleigh calibrated to COCOMO.itm* for an example at the web site listed under *Available Models*.

### Rayleigh Curve vs. Flat Staffing

In contrast to a Rayleigh curve buildup, a level-of-effort staffing may be possible for well-known and precedented problems where problems are ready for solution. An example would be a relatively simple porting between platforms of a software package

with experienced developers. Since the problem has been solved by the people before, the task can be performed by an initial large staff. The project can be planned with a nearly linear tradeoff between schedule and number of personnel (i.e. with a constant staff level, the porting will take about twice as long with one half of the staff). Another example is an in-house organic project where many people can start a project compared to the slower Rayleigh buildup.

A high-peaked buildup curve and a constant level-of-effort staffing represent different types of software project staffing. Most software development falls in between the two behaviors.

## **Integrating Modeling Perspectives**

There are connections between process concurrence and Rayleigh-curve modeling that are useful for understanding the dynamics, and collectively provide a more robust framework for modeling processes. In fact process concurrence can be used to show when and why the Rayleigh curve doesn't apply. Process concurrence provides a way to model the constraints on making work available in and between phases. The work available to perform is the same dynamic that drives the Rayleigh curve, since the staff level is proportional to the problems (or specifications) currently available to implement. S-curves result for expended effort over time (the accumulation of the staffing curve) or cumulative progress when a Rayleigh staffing shape applies.

However the Rayleigh curve was based on the initial study of hardware research and development projects that most resemble a traditional waterfall lifecycle for unprecedented software systems. Now there are a great variety of situations that it doesn't match so well. Rayleigh staffing assumptions don't hold well for COTS, reuse, architecture-first design patterns, 4th generation languages or staff-constrained situations.

The underlying assumption that an "ideal" staffing curve is proportional to the number of problems ready for solution (from a product perspective only) still carries weight. With modern methods the dynamic profile of problems ready for solution can have far different shapes than was observed when the Rayleigh curve was first applied to software. Experimentation with the external concurrence model in the previous section showed examples of this.

Iterative processes with frequent cycles or incremental development projects generally have flatter staffing profiles. Some would argue the flat profiles are the superposition of many sub-Rayleigh curves, but nevertheless the initial assumptions were based on sequential, one-pass projects.

Other situations where the Rayleigh curve doesn't apply too well are highly precededented systems for which early parallel work can take place and the project ends with a relatively flat profile, such as a heavy reuse or simple translation project, or any situation where a gradual buildup isn't necessary. Process concurrence can produce any number of dynamic profiles, and can thus be used to model more situations than the Rayleigh curve.

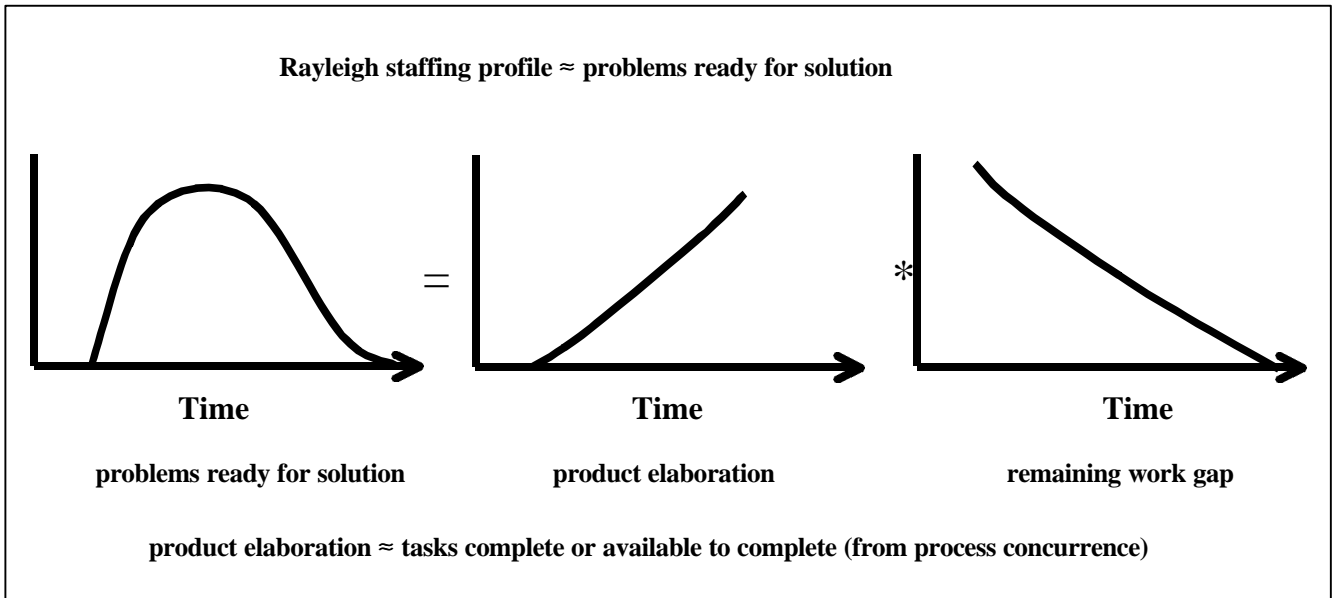
Schedule-driven projects which implement timeboxing are another example where projects that can have more uniform staffing distributions. These projects which are sometimes called Schedule As the Independent Variable (SAIV) projects since cost and quality float relative to the fixed schedule. On such projects there is no staff tapering at



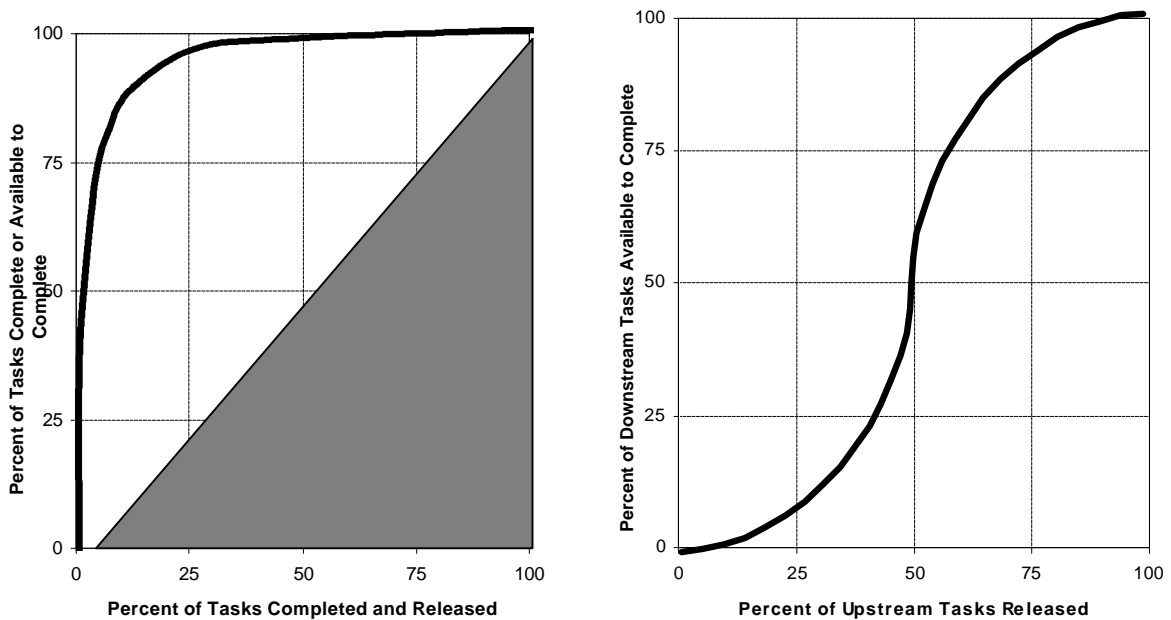
the end, because the schedule goal is attained by keeping everyone busy until the very end. Thus the staffing level remains nearly constant.

We now have alternative methods of modeling the staffing curve. A standard Rayleigh formula can be used or process concurrence can replace the elaboration function in it. The first term in the Rayleigh equation that we call the elaboration function,  $p(t)$ , represents the cumulative specifications available to be implemented. The cumulative level of specifications available is the output of a process concurrence relationship that operates over time. Thus we can substitute a process concurrence relationship in place of the elaboration function, and it will be a more general model for software processes since the Rayleigh curve doesn't adequately model all development classes. Process concurrence provides a more detailed view of the dynamics and is meaningful for planning and improvement purposes.

Recall the Rayleigh staffing profile results from the multiplication over time of the elaboration function and the remaining work left, shown in Figure 18. The elaboration function increases and the remaining work gap decreases as work is performed. Figure 18 shows the idealized Rayleigh staffing components and the process concurrence analogy for the product elaboration function. Internal and external concurrence relationships that can replace the Rayleigh formula for this are in Figure 19.



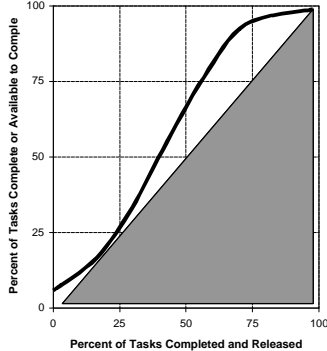
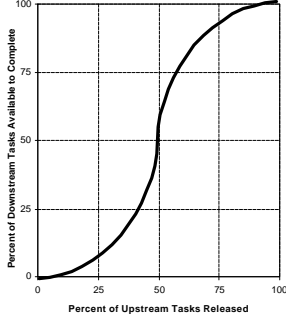
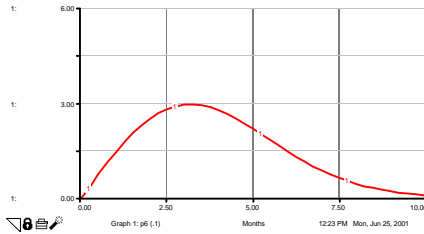
**Figure 18: Rayleigh Curve Components and Process Concurrence Analogy**



**Figure 19: Process Concurrence Replacement for Rayleigh**

Table 4 summarizes the process dynamics of tasks and personnel on prototypical projects per different modeling viewpoints that have been presented. The Rayleigh manpower buildup parameter  $a$  depends on the organizational environment and project type, so the relative differences in the table only address the effect of project type on staff buildup limits. A more precise assessment of the buildup parameter for a specific situation would take more into account.

**Table 4: Task and Effort Dynamics Summary for Major Project Types**

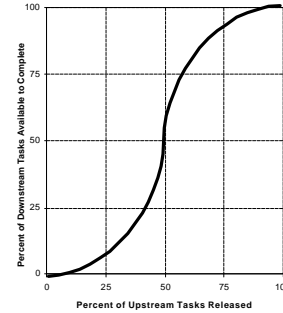
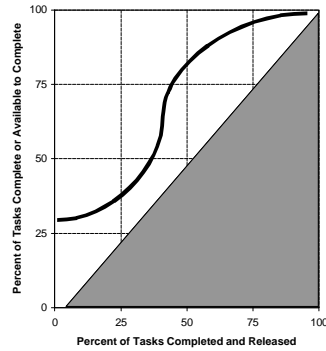
Project Type	Process Concurrence		Rayleigh Curve Modeling <sup>1</sup> ( $a$ = manpower buildup parameter)	Brooks's Interpretation
	Internal Concurrence	External Concurrence (inception to elaboration only) <sup>2</sup>		
New development of unprecedented system		 <p>initial architecture development retards design, more complete definition enables parallel development, then last pieces cause slowdown</p>	<p><math>a</math> = small</p> 	<p>Interdependent tasks impose sequentiality, so many tasks cannot be partitioned for parallel development</p> <p>∴ men and months are not interchangeable</p>

<sup>1</sup> The generalized buildup patterns shown assume that all else is held constant between these examples except for the project type. Different organizations will exhibit different buildup patterns if they had to staff the same given projects. Some organizations will always be more nimble for quick staff buildup due to their internal characteristics.

<sup>2</sup> There is often self-similarity observed in the different phases. In many instances the concurrence between elaboration and construction will be similar.

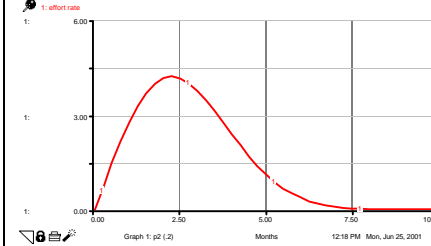
New development of  
precedented system

- internal knowledge  
of existing domain  
architectures



curve will vary with  
degree of concurrence

$\alpha$  = small or medium depending on  
organization

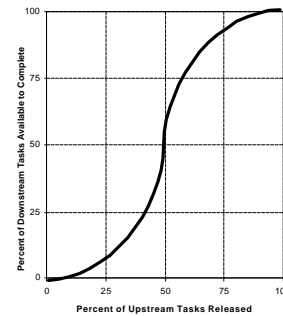
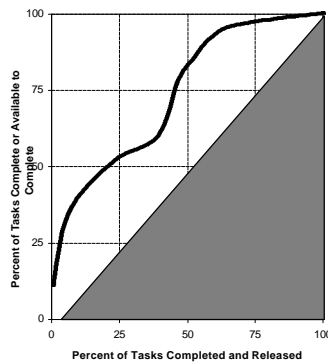


A mix of independent  
and interdependent tasks.

$\therefore$  men and months are  
partially interchangeable

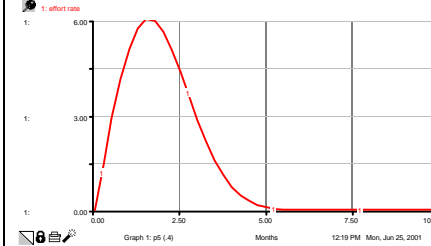
Reuse and new  
development

-bottom-up reuse  
provides some initial  
components



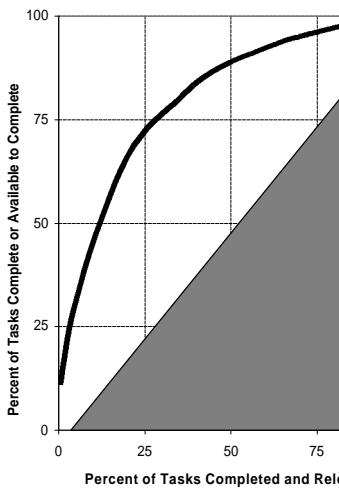
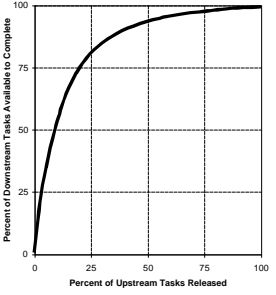
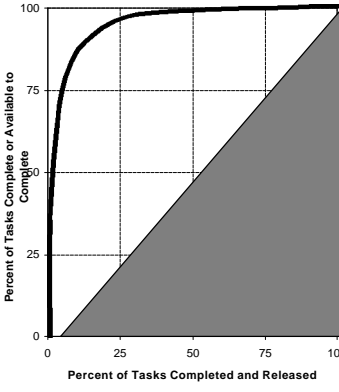
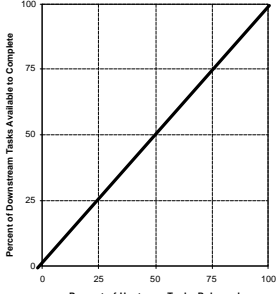
curve will vary with  
degree of concurrence

$\alpha$  = medium - large



A mix of independent  
and interdependent tasks.

$\therefore$  men and months are  
partially interchangeable

<p>COTS-based</p> <ul style="list-style-type: none"> <li>- most requirements pre-defined by COTS capabilities</li> <li>- some glue code development necessary</li> </ul>		 <p>leveraged process instantiates software from defined COTS capabilities</p>	<p><math>a = \text{large}</math></p> <ul style="list-style-type: none"> <li>- not a good fit for steady-state portions of rectangular staffing (but models extreme ramping up/down portions)</li> </ul>	<p>A mix of independent and interdependent tasks.</p> <p><math>\therefore</math> men and months are partially interchangeable</p>
<p>Translation of existing application</p> <ul style="list-style-type: none"> <li>- pieces can proceed in parallel until final integration testing</li> </ul>		 <p>elaboration pieces proceed at same rate as inception</p>	<p><math>a = \text{large}</math></p> <ul style="list-style-type: none"> <li>- not a good fit for steady-state portions of rectangular staffing (but models extreme ramping up/down portions)</li> </ul>	<p>Tasks can mostly be partitioned with no communication between them</p> <p><math>\therefore</math> men and months are largely interchangeable</p>

## Available Models

Some executable simulation models used in this work and provided to the public domain on the web at <http://rcf.usc.edu/~madachy/spd> are listed below.

- *Rayleigh* – This models the components of the Rayleigh staffing curve over time to provide understanding of process dynamics.
- *External Concurrence* – This models external process dynamics in the elaboration phase of a project and is used to derive optimal staffing profiles (from a product perspective) for different types of projects. One can explore various combinations of specification input profiles and concurrence relationships between inception and elaboration to see their effect on optimal staffing.
- *Brooks Law* – Brooks's explanations are relevant because he describes how task interdependencies constrain parallel work. This is a reference model of the Brooks's Law mechanics.
- *MBASE Architecting* - A detailed model of the MBASE architecting phase that includes internal and external concurrence between activities. This was a student term project at USC.

## Summary and Conclusions

In summary, process concurrence provides a robust framework for modeling software processes and their constraint mechanisms. It is general enough to characterize a broad spectrum of current and emerging methodologies in terms of work available to complete on a project. It can produce any number of dynamic profiles matching the different methodologies, and thus model more realistic situations than the traditional Rayleigh curve. Process concurrence can in fact be used to show when and why the Rayleigh curve doesn't apply in modern software development situations. It provides a more detailed view of the process dynamics and is meaningful for planning and improvement purposes. With it one can derive optimal staffing profiles for different project types, and as a shared project model it serves to improve stakeholder communication.

The software industry is continually introducing new processes, methodologies and tools. Process concurrence modeling can help to evaluate these new directions. Many modern techniques serve to increase concurrence in the software process in several ways. When feasible, increasing task parallelism is one of the most effective methods for achieving cycle-time reductions in software development and evolution. Leverage is also achieved by automating product elaboration. Process concurrence is ideally suited for evaluating such strategies by modeling task interdependency constraints between and within phases, and can characterize different approaches in terms of their ability to parallelize or accelerate activities.

The perspectives of process concurrence, Rayleigh curves and Brooks's Law are all related. Process concurrence models the constraints on making work available, and the work available to perform is the same dynamic that drives the Rayleigh curve since the staff level is proportional to the problems currently available to implement. Process concurrence also quantitatively illustrates Brooks's qualitative interpretation of task dependency effects. However, more empirical data is needed on concurrence relationships from the field for a variety of projects. We hope to report on this analysis in future writings.

## **Bibliography**

Brooks F, *The Mythical Man-Month*, Addison-Wesley, 1975 (also reprinted and updated in 1995)

Ford D, Sterman J, *Dynamic modeling of product development processes*, MIT D-4672, 1997

Ford D, Sterman J, *Expert knowledge elicitation to improve formal and mental models*, System Dynamics Review, Vol. 14, No. 4, 1998

Madachy R, Boehm, *Software Process Dynamics*, IEEE Computer Society, 2002 (to be published)

Putnam L, *Tutorial: Software Cost Estimating and Life-Cycle Control: Getting the Software Numbers*, IEEE Computer Society Press, New York, NY, 1980