

Overcoming the 90% Syndrome: Iteration Management in Concurrent Development Projects

David N. Ford¹ and John D. Sterman²

Abstract

Successfully implementing concurrent development to reduce cycle time has proven difficult due to unanticipated interactions of process constraints and managerial decision making. We develop a dynamic project model that explicitly models these interactions to investigate the causes of the "90% syndrome," a common form of schedule failure in concurrent development. We find that increasing concurrence and the common propensity of workers to conceal required changes from other development team members aggravate the syndrome and degrade schedule performance. We explore iteration management policies to improve performance in concurrent development projects.

1. Introduction

Developing products faster has become critical to success in many industries, whether the product is an office building, software package, or computer chip. Calls for faster product development have simultaneously taken on the sacred tone of a mantra and the volume of a brass band. Perhaps with good reason. Cycle time reduction is considered crucial to success by many researchers and practitioners with impacts larger than those of product development budget overruns. Developing products faster than competitors can increase market share, profit, and long term competitive advantage (Wheelwright and Clark 1992, Meyer 1993).

In response many firms have shifted from a sequential to a concurrent development paradigm. Concurrent development is designed to reduce cycle time primarily by overlapping development activities to perform them as much in parallel as possible. Large reductions in cycle time can be realized by applying concurrent development (Backhouse and Brookes 1996, Wheelwright and Clark 1992, Womack, Jones and Roos 1990). However, successfully implementing concurrent development has proven difficult for many organizations. Some

implementation failures and challenges have been linked to the increased coordination required among development team members by the overlapping of activities (Haddad 1996). For example Wheelwright and Clark (1992) cite a case in which applying concurrent engineering increased cycle times due to unsuccessful cross-functional teams, the primary coordinating structure in concurrent development.

Several explanations have been suggested for the concurrent development implementation challenge. Backhouse and Brookes (1996) present nine sets of contrasting case studies to support their theory that concurrent development implementation fails due to mismatches among a development organization's people, controls, tools, processes and structure, and the organization's need for efficiency, focus, incremental change, radical innovation and proficiency. Other research identifies characteristics common to most concurrent development efforts such as the disaggregation of work into smaller pieces (Smith and Eppinger 1997a). The disaggregation of development activities into smaller tasks for increased concurrence increases the number of tasks begun with incomplete or preliminary information, boosting the number of design iterations. As a result concurrent methods increase the frequency and magnitude of information transfers. Therefore cycle time reduction through concurrent development comes at the cost of increased process and management complexity (Wheelwright and Clark 1992, Krishnan, Eppinger and Whitney 1995). Development process design and management policies have not generally improved to address the effects of increased complexity due to concurrent development. This has contributed to the frequently cited poor management of development projects (Womack et al. 1990) and failure to meet project goals.

2. The 90% Syndrome

One common concurrent development problem is the "90% syndrome." The syndrome describes a project that reaches about 90% completion according to the original project schedule but then stalls, finally finishing after about twice the original project duration has elapsed. The phenomenon is so common many design engineers have codified it into their

normal expectations. A product development manager in one company we worked with described their experience as "The average time to develop a product is 225% of the projected time, with a standard deviation of 85%. We can tell you how long it will take, plus or minus a project." (Ford, Hou, Seville 1993). The syndrome is common in many industries including software, construction, consumer electronics and semiconductors. The management of the final 10% during the last half of the project is typically of the most concern to managers because this portion of the project obviously and significantly deviates from planned progress, focusing attention on the failure of the project to meet its targets. Our fieldwork suggests inter-phase iteration in the latter half of the project and the late discovery of unanticipated rework as keys to understanding the syndrome.

Eliminating the 90% syndrome could potentially reduce cycle time roughly 50%. In light of the relatively recent or slow adoption of concurrent development in many industries the syndrome could be assumed to be a temporary problem that will disappear as firms become competent at managing concurrent development. However the persistence of the syndrome in construction projects several decades after the widespread adoption of overlapping phases (fast tracking) and cross functional development teams suggests that the syndrome is a persistent concurrent development problem and not a transient issue soon overcome by experience.

Effective process and policy design requires explaining how the structure and characteristics of projects and the behavior of the participants can cause the 90% syndrome, particularly when concurrent development practices are being used. Many existing models treat either the structure of development processes without regard to behavioral issues or focus on behavior without considering the process structures. Abdel-Hamid (1988) used a simulation-based approach, integrating process and behavioral issues to investigate the 90% syndrome in software development projects. He found the cause of the syndrome to be the interaction of underestimating project size and poor assessments of true project progress (that is, the inability to detect design flaws until work reaches a downstream testing phase). Abdel-

Hamid identifies one development process (delays in error detection) and one management decision (estimating project size) as critical elements in the 90% syndrome. But the theory doesn't apply to projects with clear and stable scopes that nevertheless experience the syndrome. Building on this work, we develop a more general explanation for the syndrome in projects, integrating physical and information processes with managerial decision making.

In this article we address the need for better understanding of effective concurrent development implementation by showing how processes and policies can interact to degrade schedule performance in concurrent development projects. We use a causal dynamic model to explore interactions of concurrent process structure with the behavioral decision processes of the actors in the system. In particular, we explore the impacts of the interaction of process concurrence with the common behavior of concealing the need for rework on schedule performance. We show how these interactions can cause the "90% syndrome," a chronic form of schedule failure and use our results to suggest policies for reducing cycle time.

3. The Development Project Model

To investigate the impacts of the interaction of physical and information processes with managerial decision making we built a dynamic project simulation. The model is a system of nonlinear differential equations describing (i) the flow of work and information within and among development project phases, and (ii) the policies used to manage the project. Model components and their interactions are based on existing product development theories (Smith and Morrow, forthcoming) and our field studies of development projects. For example the development process structure is based on theories of project constraints and resources (Noreen, Smith and Mackey 1995) and previous dynamic project models including Abdel-Hamid (1988), Ford and Serman (1998a), Cooper (1980) and others. Because no closed-form solutions are known we simulate the system's behavior.

Our model simulates the performance of a multiple-phase development project. Each phase is represented by a generic structure, which is customized through parameterization to reflect a

specific stage of product development such as preparing construction drawings for building an office complex, writing software or testing computer chip prototypes. The numeraire for development work is the "task," an atomic unit of work. Examples of development tasks include the selection of a pump, writing a line of code or installing a steel beam. When tasks within a phase are heterogeneous the unit of work can be defined as the average amount an experienced person can accomplish in a given time interval (e.g. an hour or day). Information concerning the quality of completed tasks is generated through testing or checking through quality assurance efforts. Tasks may require changes because they were done incorrectly or because the information they were based on has changed. We refer to this development activity as rework. Consistent with Safoutin and Smith's (1998) use of the term "modifications", rework is changes to generate needed information, correct errors (mandatory) or improve performance (optional) and can be intentional or unplanned. We assume tasks are small enough to either require a change or be correct and in final form, but not to require a partial change.

Modeling the Flow of Development Work

The flows of work and information among phases define the network structure of the project. Figure 1 shows a simple but common example. The links shown in Figure 1 represent several forms of inter-phase interaction, including:

- Progress in receiving phases is constrained by supplying phases that provide them with development products or other information. These flows are shown by the solid arrows.
- Work inherited by receiving phases from supplying phases may require rework. Inheritance of tasks requiring changes causes work done by the receiving phases to also require changes. When corrupted work is discovered it is reported to the phase where the change requirement was generated so it can be reworked. These flows are shown by the dashed arrows in the project network.
- Rework requires coordination between the phase that discovered the change requirement and the phase that generated and must correct the work. This coordination must occur prior to the revision of the work.

Consistent with the literature on concurrent development (e.g., Terwiesch, Loch and DeMeyer 1998, Smith and Eppinger 1997a,b) the information flows among phases in the model are bi-directional. Safoutin and Smith (1998) describe the inter-phase iteration we focus on as meso in scale to distinguish it from iteration within individual phases (micro) or between product releases (macro). Figure 1 also shows the Design Structure (Smith and Eppinger 1997a,b) corresponding to the project network in the diagram in which several iterative loops are created by the bi-directional information dependencies among phases.

All development processes are constrained by the physical and information relationships among the tasks and phases within a project. These constraints include traditional task durations and precedence relationships but also include information dependencies leading to iteration (Smith and Eppinger 1997b). Managerial decisions about the availability of work, coordination mechanisms (Hauptman and Hirji 1996) and the characteristics of information transferred among development phases (Krishnan 1996) can also affect progress, as can the number, skill, and experience of project staff (Abdel-Hamid 1988). These processes and policies can interact to constrain progress. Consider as an example the erection of the structural steel skeleton for a ten-story building. Steel erection requires that the structural members (the columns, beams and bracing) be installed, inspected for proper installation and corrected if the installation is found to be defective. For each member these activities can only occur in a specific order: install, inspect, approve or discover an error and correct, re-inspect rework, and eventually release to other development phases. Management policies such as the number of floors of steel that must be approved prior to being released also affect progress and information availability downstream. Figure 2 shows the storage and flow of work within a single development phase such as the steel erection example.

Basework

In our model any task is in one of six states represented by boxes in Figure 2: base work not completed (W_b), work completed but not yet checked by quality assurance (W_q), work that has been approved (W_a), work which has been finished and released to other development phases or customers (W_f), work requiring coordination between development phases prior to being changed (W_c) and work known to need changes and awaiting rework (W_r). Four fundamental development activities drive the movement of work among these six stocks. Base work (b) is the completion of work for the first time and moves work from its initial condition of not completed to being completed and needing checking. Quality assurance (q) is checking for errors and improvement needs. Tasks that do not require rework or tasks with changes not discovered during quality assurance are approved (a) and accumulate in the stock of work approved until they are released (f) to the managers of receiving phases or to customers. Quality assurance can also discover work needing changes (d) due to flaws by developers in the phase or the inheritance of defective, incomplete or preliminary work from supplying phases. Rework can be generated in supplying phases by changes in customer requirements or other environmental disruptions. Rework (r) seeks to correct errors or improve tasks previously completed. Work that has been corrected or improved (i.e. reworked) is returned to the completed-not-checked stock for re-inspection. Thus:

$$(d/dt) W_b = - b \quad (1)$$

$$(d/dt) W_q = b + r - d - a \quad (2)$$

$$(d/dt) W_a = a - f \quad (3)$$

where: W_b - Work needing Base work W_a - Work Approved
 W_q - Work needing Quality Assurance a - Approval rate
b - Base work rate d - Discovery of rework rate
r - Rework rate f - Finish and release rate

Discovery and Notification of Rework Requirements

Recognizing and reworking errors is critical to product development success. In our model work discovered to need changes moves from the stock of work awaiting inspection to the

stock of work to be coordinated (Figure 2). Coordination (c) is the integration of the project among phases. Classic examples include designers working with marketers to refine product specifications, or engineers meeting with product architects to explain why certain product specifications cannot be met within the time available. We assume a fortiori that coordination across phases can be accomplished asynchronously with tools such as memoranda, sharing of CAD/CAM files or electronic mail. After coordination resolves any disputed issues tasks move to the stock of work to be changed and are subsequently reworked. Work requiring changes can also be missed during quality assurance and therefore be approved and released, to be discovered in a receiving phase. Upon discovery the generating phase is notified and the affected work is moved from the stock of work considered finished (W_f) to the stock of work to be coordinated (W_c). After coordination, the change is made by the originating phase and the work is checked and eventually released again. These flows yield:

$$(d/dt) W_f = f - n \quad (4)$$

$$(d/dt) W_r = c - r \quad (5)$$

$$(d/dt) W_c = d + n - c \quad (6)$$

where: W_f - Work Finished and released W_r - Work known to need Rework
 W_c - Work needing Coordination c - Coordination rate
 n - Rate of Notification and return of work requiring coordination and
rework

The detection of tasks requiring rework is not a purely technical matter. There are often long delays between the time receiving phases detect a problem requiring rework and the notification of the phase where the problem was generated. Engineers in a phase often have incentives to hide errors from management and from other phases, for example to maintain the impression that their phase is progressing on schedule. They hope others will be forced to reveal their errors first, causing a schedule slip that enables them to solve the problem before having to reveal it. We capture these behavioral processes by distinguishing between the detection of tasks requiring rework (d) and the notification of affected phases (n).

Quality Assurance

Three of the four fundamental development activities (base work, rework and coordination) are used directly in equations (1), (2), (5), and (6). The discovery of changes and approval of work are based on the fourth fundamental development activity: quality assurance. We separate the work inspected through quality assurance (q) into the discovery of generated changes (d) and the approval of work (a) as follows. Since quality assurance efforts are not perfect, the need for some rework is missed. Some work that must be changed is mistakenly considered to be finished, and is released (type I errors). We assume that the work done correctly is not mistakenly flagged for rework (no type II errors), though the model can easily be expanded to relax this assumption. Therefore the fraction of the work under-going quality assurance discovered to require rework is the product of the probability that a task requires a change ($p(\)$) and the accuracy of quality assurance, which we measure with the probability of discovering the need for a change given that the need exists ($p(\ | \)$). The first probability is strongly influenced by product complexity but also by process complexity. The sensitivity to quality assurance ($p(\ | \)$) is affected by managerial decisions that influence the quality of inspection such as the experience of the quality assurance work force and by managerial objectives and incentives to reveal or conceal change requirements. The fraction of tasks discovered to require rework is the non-overlapping sum of the probability that a task requires rework due to causes within the phase itself ($p(\ i)$) and the probability that a task requires rework due to problems inherited from its supplier phases ($p(\ e)$). Tasks for which no rework needs are discovered are approved.

$$d = q * p(\) * (p(\ | \)) \quad (7)$$

$$p(\) = (p(\ i) + p(\ e)) - (p(\ i) * p(\ e)) \quad (8)$$

$$a = q - d \quad (9)$$

where: q - Quality assurance rate

$p(\)$ - probability a task requires rework

$p(\ | \)$ - probability of discovering the need for rework if it exists

$p(\ i)$ - probability a task requires rework due to internal causes

$p(\ e)$ - probability a task requires rework due to external causes

While work is approved continuously, the release of approved work (f) is discrete. Approved work (W_a) is released in packages of size r . The release package size is a management decision and is strongly conditioned by characteristics of the phase. For example in computer chip development the vast majority of the design code must be completed prior to release for a prototype build since almost all the code is needed to design the masks. Likewise, the release of construction documents for public bidding is contingent on the completion of a majority of design work. In other development settings managers have broad discretion in setting release package sizes. Product specifications can be released to design a few specifications at a time or in a single complete document.

One release policy uses the release package size to specify the minimum fraction of the unreleased work (the phase scope, S , less the work released, W_f) that must be approved to begin releasing work. The phase scope is the total number of tasks to be done in the phase. Using this policy the release rate is zero when the stock of approved work is less than the threshold release packet size. Once the stock of approved work exceeds the threshold it is released over a period determined by the nominal release time (r). The nominal release time can include physical and administrative processing delays, can vary across projects or phases, and is influenced by phase managers. We assume the instantaneous release of tasks approved to date by setting the nominal release time to the simulation time step. This policy releases work in packages of decreasing size.

$$f = \begin{cases} W_a / r & \text{if } W_a \geq r * (S - W_f) \\ 0 & \text{otherwise} \end{cases} \quad (10a)$$

where: r - release package size as fraction of unreleased work

S - phase scope

r - nominal release time

An alternative policy releases work in packets of fixed size r (the last packet may be smaller).

$$f = \begin{cases} W_a / r & \text{if } W_a \geq \text{Min}(r * S, S - W_f) \\ 0 & \text{otherwise} \end{cases} \quad (10b)$$

where: r - release package size as fraction of phase scope

We use these two release policies in our model to describe the practices we have observed in the field. Other release policies can also be modeled.

The rate at which a supplying phase removes work from its stock of finished and released tasks for coordination and rework (n) is driven by the discovery of the need for rework by receiving phases. This flow in the supplying phase is the sum of the rework requirements discovered by all receiving phases. The discovery of rework in receiving phases due to using supplied work that need changes depends on the probability that the work released by a supplier phase, s , to a receiving phase, r , requires a change ($W_{f,s} / W_{f,s}$ where indicates required rework), the rate at which the receiving phase checks work (q_r) and the probability the receiving phase discovers the inherited change ($p(\cdot | \cdot)_r$). This rework discovered in receiving phases is scaled for differences in phase scopes (S_s/S_r) to describe the rework required in the supplying phase. The contribution of returned work needing rework to a phase is zero from all phases that do not utilize the phase's output because no work is released to these phases. Simultaneous discovery of the same inherited change requirement by multiple receiving phases is assumed insignificant. This assumption is valid for relatively small change densities and a sufficiently short time step in the simulations. Work requiring changes is modeled with two structures that have the same basic stock and flow structure shown in Figure 2, one each for changes due to internal and external causes.

$$n_s = \sum_r [(W_{f,s} / W_{f,s}) * q_r * (p(\cdot | \cdot)_r) * (S_s / S_r)] \text{ for } r \in \{1,2,\dots,m\} \text{ and } s = r \quad (11)$$

where: $W_{f,s}$ - Work requiring rework but released by supplying phase s

$W_{f,s}$ - Work released by supplying phase s m - number of phases

Modeling Concurrency and the Availability of Work

In general, the rate at which any of the development activities (base work, quality assurance, coordination, and rework) is performed depends on two types of constraints. First, work rates can be constrained by inadequate resources – too few workers, insufficient worker skill and experience, or insufficient supporting infrastructure (such as CAD/CAM systems). Second, progress can be constrained by the interdependencies among phases and tasks. As an example, consider the erection of the structural steel skeleton for a ten-story building. Each member must be installed (base work), and inspected (quality assurance). If an error is found, there must be coordination among the affected supervisors and skilled trades (coordination) before the error can be corrected (rework). For any given technology, a certain minimum amount of time is required for each of these four activities to be performed for each structural member even when resources such as laborers and cranes are ample. Further, certain tasks cannot be worked on until others are done.

The rate of progress $p_{j,k}$ for activity k in phase j is the minimum of (i) the rate feasible due to the quantity and productivity of resources allocated to each activity and phase ($p_{l,j,k}$ where l refers to labor as an example of a resource), and (ii) the rate feasible given the number of tasks available to the activity ($w_{j,k}$) and the minimum activity duration for that activity ($t_{j,k}$):

$$p_{j,k} = \text{Min} (p_{l,j,k} , w_{j,k} / t_{j,k}) \quad \text{for } j \in \{1,2,\dots,m\} \text{ and } k \in \{b, q, c, r\} \quad (12)$$

where: $p_{j,k}$ - Progress feasible for activity k in phase j

$p_{l,j,k}$ - Progress feasible due to resource constraint l for activity k in phase j

$w_{j,k}$ - Work available for activity k in phase j

$t_{j,k}$ - Minimum activity duration for activity k in phase j

It is obvious that inadequate or poorly trained workers, or shortages of required equipment and support infrastructure, can cause delays in projects and may contribute to the 90% syndrome. A variety of models explore how underestimation of project scope, overestimation of productivity, or unexpected changes in customer requirements can lead to delays and cost overruns by causing resource shortages (e.g. Abdel-Hamid 1988). In this paper, however, we seek to show how the structure of a development process and managerial decision making

interact to contribute to the 90% syndrome, even when resources are ample. Thus we assume henceforth that resources are always more than sufficient, and equation (12) reduces to:

$$p_{j,k} = w_{j,k} / j_{,k} \quad \text{for } j \in \{1,2,\dots,m\} \text{ and } k \in \{b, q, c, r\} \quad (12a)$$

The minimum activity durations $j_{,k}$ are specific to each activity and phase. These durations must be estimated empirically, for example from time and motion studies, the history of past projects, or other means. In the structural steel example the minimum base work duration ($j_{,b}$) would answer the question "How long does it take fully experienced people with ample and appropriate materials, equipment and information to place a steel member?"

We model the work available ($w_{j,k}$) for quality assurance, coordination, and rework with the stocks of tasks awaiting each of these activities. Thus:

$$w_{j,k} = W_{j,k} \quad \text{for } k \in \{q, c, r\} \quad (13a)$$

The formulation for base work available is different. The base work available, $w_{j,b}$, is all the work that can be initially completed less the work that has already been completed. Base work tasks already completed is the sum of all tasks that have been released, approved, are awaiting rework, or awaiting quality assurance, or, equivalently, the difference between the phase scope, S_j , and the tasks remaining in the base work stock, $W_{b,j}$. Thus base work available for initial completion is given by:

$$w_{j,b} = \text{Max}(0, w_{j,t} - (S_j - W_{b,j})) \quad \text{for } j \in \{1,2,\dots,m\} \quad (13b)$$

where: $w_{j,t}$ - Total work available for base work completion in phase j

The Max() function is required since the total number of tasks available for base work whether completed or not ($w_{j,t}$) depends on the tasks released to phase j by the phases that supply it with development information. Suppose supplying phases have released a large fraction of their tasks, so that phase j is able to complete, say, half of its work (that is, $w_{j,t} = .50*S_j$). Now, however, suppose a large number of errors are discovered in the work of the supplying phases. The tasks previously released to phase j are recalled for rework and the number of tasks that phase j can do ($w_{j,t}$) falls (e.g. to $w_{j,t} = 0.25*S_j$) until the flawed tasks are coordinated, reworked, re-inspected, approved and re-released by the supplying phases. If

phase j had already completed most of the tasks that depended on the flawed work (e.g. $S_j - W_{j,b} = 0.45 * S_j$) the term $w_{j,t} - (S_j - W_{j,b})$ becomes negative. The $\text{Max}()$ function enforces the non negativity constraint on work available in situations where a phase has done more work than it should have given the tasks released by the phases upon which it depends. In such circumstances, further base work in phase j must cease until the flawed tasks upon which it depends are reworked and re-released. In addition, of course, the base work completed by phase j based on the flawed work will also require rework. For example the steel erection phase could be forced to stop installing new members if released drawings of installed work were recalled for design changes. Upon the re-release of the drawings the redesigned members already installed could be corrected.

Another constraint on the availability of work arises from the fact that not all tasks can be developed fully concurrently. The progress of work within a phase can constrain the availability of work within that same phase. For example in the steel erection phase of a building project, work on some steel members such as beams must wait for the completion of other work such as the installation of the columns that support those beams, and the upper floors are not available until the lower floors are completed. Therefore not all the structural members can be installed simultaneously. The progress of work in a phase can also be constrained by the completion and release of work by the phases that supply it with development products and information. The number of tasks available is the minimum of the internal constraints set by the focal phase itself (a_j) and the external constraints set by its supplying phases ($a_{s,j}$).

$$w_{j,t} = S_k * \text{Min}(a_j, a_{s,j}) \quad \text{for } s \text{ and } j \in \{1,2,\dots,m\} \quad (14)$$

where: a_j - fraction of phase j scope available due to internal dependencies

$a_{s,j}$ - fraction of phase j scope available due to external dependencies with supplying phase s

The work availability constraints, a_j and $a_{s,j}$, are based on the amount of work that has been completed to date and therefore limit the degree to which tasks can be done concurrently. We model these constraints with concurrence relationships. These functions describe the fraction

of the phase scope available for initial completion as it depends on the fraction of required information available. The relationships answer the question "How much work can now be completed based upon how the work has progressed thus far?" We describe work availability constraints imposed by a phase on itself with the internal concurrence relationship (α_j) and the constraints imposed on a phase by supplying phases with external concurrence relationships ($\alpha_{s,j}$ where s represents a supplying phase and j the receiving phase). The fraction of tasks available as the basis for additional work due to an internal concurrence constraint (α_j) is a function of the fraction of the phase scope that is perceived to have been completed satisfactorily, i.e. the work project managers expect do not require changes. The fraction of tasks perceived satisfactory is the ratio of the work perceived satisfactory thus far and the phase scope. The number of tasks perceived to be completed satisfactorily is partially a managerial decision. It includes at least the sum of the work approved ($W_{j,a}$) and the work finished and released ($W_{j,f}$). In addition development managers may decide to also include the work completed but not yet checked by quality assurance ($W_{j,q}$) in the work perceived to be satisfactory. The impacts of this policy can be investigated with our model. We include unchecked work based on the results of our field studies where we found, for example, that programmers continued to write code based on prior code even when that software had not been tested and approved. Tasks known to need rework or awaiting coordination are not included because they do not make additional work available.

$$\alpha_j = f_j((W_{j,q} + W_{j,a} + W_{j,f}) / S_j) \quad (15)$$

where: α_j - concurrence relationship of phase j

Internal concurrence relationships capture the degree of sequentiality or concurrence of the tasks aggregated together within a phase, including possible changes in the degree of concurrence as the work progresses. The internal concurrence relationships characterize the dependencies represented by the diagonal terms in the Design Structure Matrix. To prevent describing conditions in which work can be completed before only after it has already been completed the fraction of tasks available as the basis for additional work (α_j) must be greater than the fraction of tasks perceived satisfactory ($(W_{j,q} + W_{j,a} + W_{j,f}) / S_j$). Alternatively, a

graph of a phase's internal concurrence relationship (a_j) must lie strictly above a 45° line. Within the feasible region a variety of functional forms are possible including nonlinear concurrence relationships. For example consider the internal concurrence relationship for the steel erection of a ten-story office building in which the floors are erected sequentially from the ground up. At the beginning of the project only the first floor (10%) is available to be completed ($(W_{j,q}+W_{j,a}+W_{j,f})/S_j=0\%$ and $a_j=10\%$). The completion of portions of the first floor make available portions of the second floor. When the first floor is finished, 10% is completed and the second floor is available, making 20% available for initial completion ($(W_{j,q}+W_{j,a}+W_{j,f})/S_j=10\%$ and $a_j=20\%$). This linear progression continues ($a_j=10\%+((W_{j,q}+W_{j,a}+W_{j,f})/S_j)$) until the completion of the ninth floor releases the final floor for completion. In general, more concurrent processes are described by curves near the upper and left axes of the internal concurrence graph and less concurrent processes are described by curves near the 45° line.

External concurrence relationships describe the available work constraints between development phases in an analogous manner. Testing, for example, cannot commence until a prototype is completed. An external concurrence relationship describes the fraction of work that can be done in a receiving phase based on the fraction of work released by a supplying phase. The external concurrence relationships characterize the interphase dependencies represented by the off-diagonal terms in the Design Structure Matrix. They are potentially nonlinear, allowing our model to capture changes in the degree of dependence among phases as a project evolves. The fraction of tasks released from the supplying phase is the ratio of the number of tasks released by that phase ($W_{s,f}$) to the scope of the phase (S_s). The function is identically unity for all phases s that do not supply information to phase j .

$$a_{s,j} = f_{s,j} (W_{s,f} / S_s) \quad (16)$$

where: $f_{s,j}$ - concurrence between supplying phase s and receiving phase j

Concurrence relationships are characteristic features of a project's network structure and must be estimated for each project. We tested our model against the behavior of a medium-sized

chip development project at a major U.S. semiconductor firm (the Python project). Ford and Sterman (1998b) describe the protocol used to elicit these concurrence relationships and provide examples. These workshops proved to be useful not only for model calibration but also helped the members of the development team to understand each other's perspectives of the development process.

As an example Figure 3 shows four expert estimates of the external concurrence relationship between the product definition and design phases of the Python project. The product definition phase develops product architecture and specifications based on the Python chip's market and target performance. The designers use these specifications as the basis for the detailed design embodied in the software code used to lay out the chip's individual components. Each estimate in Figure 3 describes the mental model of a participant in the product definition or design phase concerning the question "How much design work can be completed based on the fraction of the product definition work that has been completed, approved and released?" Interestingly, the two "upstream" people (the marketing representative and the product architect) believed the "downstream" people (the design manager and designer) could and presumably should begin their work quite early, when few product specifications have been released, while those downstream believed their work could only progress when a majority of the specifications were released. These differences in mental models had led to conflict and delay in prior development projects. The explicit description of these mental models initiated and facilitated discussions for improving the organization's development processes.

Equations (1) through (16) describe a model that captures both process and managerial constraints. Our model is generic in that it can describe many development phases and can represent, through suitable parameterization, how physical and information development processes and managerial policies constrain and propel work through different development phases and control the quantity or timing of rework.

4. Concurrency-generated 90% Syndrome

We used the model to test whether the information interdependencies created by concurrent development alone can cause the 90% syndrome. The Python project applied all the major components of concurrent development including overlapping phases and cross-functional teams. The organization was well-trained in concurrent development practices (Ford 1995). Figure 4 compares the Python project's original schedule and actual performance, developed from records of the phase durations and completion dates. As is common in the development process for semiconductor chips, and verified in our interviews with Python developers and managers, the design phase planned to release its work in a single large package, generating the vertical jump in planned progress. However after development began the decision was made to design the Python chip in two components instead of one. This resulted in two releases from design and therefore the two jumps in actual performance. The Python project suffered from the 90% syndrome. The project remained close to the original schedule through week 20 and was 73% complete by the original deadline. But progress then slowed from 1.8% per week to 0.9% per week, and the project was ultimately completed 77% late (week 69 versus 39).

We calibrated our model to the Python project. Parameter estimates are based on the literature and our field studies. Figure 5 shows planned and actual project performance as simulated by our model. Planned progress simulates management's plan for a single design release, their assumption of no iteration, and overestimation of productivity. Experienced managers expect iteration (lower net productivity) but are often required to use "stretch objectives" in project planning in the belief they keep motivation and pressures to perform high. Stretch objectives are supported by the literature and our field data in which Python project developers repeatedly described the unrealistic optimism used to plan projects, including the assumption of few or no changes. Plans assuming no iteration can cause problems in the allocation of available resources between iteration activities and planned activities and in reduced productivity.

The model recreates the 90% syndrome experienced by the Python project. Differences between our simulation and the project's actual behavior are largely due to resource impacts omitted from the model, specifically staffing problems created by the unanticipated iterations. The similarity in behavior patterns between Figures 4 and 5 indicates that the informational and physical dependencies created by concurrence can generate the syndrome and cause projects to take longer than planned. This suggests that even projects managed by highly skilled project managers with adequate resources can experience the 90% syndrome.

Of particular importance to implementing concurrent development is the impact of concurrence on schedule performance. Figure 6 shows the simulated Python project with: (1) the degree of concurrence actually used, (2) a process in which all concurrence relationships are 25% more overlapped than actual conditions, and (3) a process in which all concurrence relationships are 25% less overlapped. In describing increased and decreased concurrence we retained the basic shapes of the relationships, effectively shifting them left to increase concurrence and right to decrease concurrence. The traditional perspective suggests more concurrence should help projects finish significantly earlier, while less should cause projects to finish later. Figure 6 shows that decreasing concurrence delays the project 34 weeks (49%) but increasing concurrence by the same amount improves schedule performance by only 7%.

The weak response to enhanced concurrence and the persistence of the 90% syndrome arises from unplanned iteration cycles to correct unanticipated errors. In an iteration cycle, the discovery of a change is passed from the developers or managers in the discovering phase to those in the originating phase and the rework required is coordinated among the affected phases. Changes are made to the flawed tasks in the originating phase and to contaminated work in all affected phases. The originally flawed work is re-inspected, re-released and arrives at the location of its discovery again. For example, a test phase may discover an error in the chip such as a short circuit across two layers. Suppose the error is traced to the design. Test engineers notify and coordinate with the designers to specify the location and characteristics of the short circuit. The designers then must rework, recheck and re-release

the design, followed by changes in layout, tape out, masking, and prototype fabrication. The iteration cycle is completed when testing of the redesigned prototype begins. Figure 1 and equations (2) - (11) with the appropriate subscripts define this path in the model.

The impact of concurrence on schedule performance is asymmetric because the effects of iteration cycles on schedule performance increase as projects become more concurrent. As development phases are performed more concurrently more work is performed before rework requirements due to change requirements in other phases are discovered. Consequently, errors generate additional rework and iteration cycles in more concurrent projects, offsetting much of the schedule performance improvement gained from increased concurrence. Increased concurrence also increases the average iteration path length by delaying the discovery of the need for rework to phases farther from the generating phase.

5. Concealing Rework Requirements in Concurrent Development

Our field studies of concurrent development projects indicate managers and developers behave in ways that exacerbate the 90% syndrome through the information processing structure described above. Here we focus on one common and problematic behavior: purposefully concealing change requirements from development team members. We have found evidence of concealing change requirements in many development projects. Concealment can be used to temporarily reduce work. For example an engineer in a leading electronics company reported to us that design engineers regularly delayed revealing problems they discovered to avoid time-consuming document control work required by the organization's engineering change notice process (Ford et al. 1993). However the practice of concealing rework requirements is often more systemic due to people's dislike of bad news and information that contradicts their beliefs. People in authority often "shoot the messenger" - punishing those who convey bad news. Consequently, people suppress information they believe will be unpleasant to their superiors or important customers. For example suppliers are reticent to report that parts will be late even if they are integral members of their customer's development team (Roth and Kleiner 1996, p. 43). The practice of hiding one's

mistakes is institutionalized in many organizations. A manager at a major automobile manufacturer we refer to as AutoCo observed the pervasiveness of the practice of concealing discovered rework in the organization: "There is a basic cultural commandment in engineering - don't tell someone you have a problem unless you have the solution. You're supposed to solve it - and then tell them." (Roth and Kleiner 1996, p. 13-14)

Concealment can be standard practice for entire projects. At a major defense contractor, weekly meetings of project team leaders were known as "the liars' club" because everyone withheld knowledge that their part of the project was behind schedule. Members of the liar's club concealed development problems in the hope that someone else would be forced to admit problems first, forcing the schedule to slip and letting them escape responsibility for their own part's tardiness. Everyone in the liar's club knew that everyone was concealing rework requirements and everyone knew that those best able to hide their problems could escape responsibility for the project failing to meet its targets.

Purposefully concealing changes from managers and other team members is counterintuitive in a concurrent development environment. The team concept of concurrent development is designed to promote open and early sharing of problems. But our fieldwork shows that this is often not practiced, even in organizations staffed with competent, well-intentioned developers and managers trained in concurrent development. What causes these managers to conceal change requirements? Concealing rework requirements provides the manager of an individual phase several benefits that make it an attractive policy beyond the avoidance of responsibility for poor project performance cited above:

- Iteration is temporarily reduced in concealing phases, increasing the amount of work completed and available to receiving phases. This improves apparent schedule performance, sometimes allowing managers to meet a critical phase deadline they would miss if change requirements were revealed. Concealment also reduces the starvation of receiving phases and pressure for information from development team members.

- Reducing current iteration reduces the number of tasks acknowledged to need coordination and rework and therefore the amount of work considered a problem or hindrance to progress. This improves apparent quality performance.
- Problems that threaten the project schedule can be hidden and resolved during schedule extensions blamed on other phase's rework. This reduces the blame for poor phase schedule performance the concealing manager must bear.
- Maintaining adequate apparent progress reduces the likelihood of upper management intervening or replacing the manager with someone believed to be better able to meet phase and project targets. Concealment enhances the manager's job security and authority.

It is tempting to believe that managers with better training or more experience in concurrent development would not conceal change requirements by joining a liar's club. But consider the predicament of managers of individual phases. They have committed themselves to complete their phase by a given deadline. Aggressive deadlines and overly optimistic assumptions about productivity and rework generate pressure to show regular progress toward these deadlines. The pressure on managers of individual phases to show progress toward their deadlines can be enormous. A manager at a major automobile maker said "...the only thing they shoot you for is missing product launch...everything else is negotiable" (Repenning and Sterman forthcoming). But the manager's phase has inherited flawed or incomplete work from its supplying phases and has also generated rework due to internal causes, increasing the pressure. Managers have several options once they realize that required rework will cause their phase to finish after the deadline. First, they can identify their phase's problems to development team members and superiors and bear responsibility for delaying the project. By doing so they expose themselves to the possibility of looking bad twice, once now for delaying the project and again later if inherited changes or their own phase's errors delay the phase again. Second, they can identify the errors their phase has inherited from supplying phases but conceal the rework generated by their own phase, blaming their peers for the problems their phase is experiencing. If this results in schedule slip, the managers have gained time to solve the problems generated by their own phase without having to take

responsibility for them. But they have certainly alienated the managers of the supplying phases they exposed and opened the door to having their phase's errors exposed by managers of receiving phases eager for another deadline extension so they can safely resolve their own problems. Finally, managers can join the liar's club by remaining silent about both their own problems and those generated by other phases, claiming adequate progress and hoping that someone else will be forced to confess and bear the wrath of superiors, providing time to solve their own phase's problems in relative secret. Only by joining the liar's club can managers avoid responsibility for failure, prevent retaliation by other managers and potentially complete their own rework before it is exposed.

When viewed from the perspective of the manager of an individual phase a policy of concealment is often rational in the short run. But inter-phase dependencies can cause major problems at the project level, leading to a tragedy of the commons. Solving problems without the assistance of development team members from other phases can be impossible in the development of complex products such as high rise office buildings or automobiles. Early in one AutoCo project the sum of the electric power demand by individual systems (starter, headlights, etc.) exceeded the maximum output of the vehicle's alternator by a factor of two. The problem could not be recognized and solved until all system groups were aware of the problem and coordinated their actions (Roth and Kleiner 1996, p 38). The agency problems motivating concealment can also create a form of prisoner's dilemma. Eventually the errors causing delays are discovered and exposed. So the liar's club is doomed. At these times liar's club members may try to reveal the flaws of other phases anonymously to gain time and shift blame but escape retaliation. An AutoCo developer reported that "...the supplier will say things like (for example), 'You didn't hear it from me, but something is going to be late and somebody's lying to you. Don't tell anyone I told you.'" (Roth and Kleiner 1996, pp. 18-19). When receiving phases eventually acknowledge inherited work requiring changes as the cause of their schedule failure they initiate iteration cycles in which the generating and intermediate concealing phases must coordinate and correct this work. Project performance then pays the cost of the temporary benefits received by individual phase managers as the

process-constrained iteration cycles degrade project performance. The greater the delays in the revelation of rework requirements, the greater the costs of remediation.

6. Modeling the Concealment of Discovered Rework Requirements

We use our concurrent development project model to investigate the effects of concealing rework requirements on schedule performance by disaggregating the fraction of work requiring rework discovered to need changes ($p(\cdot|\cdot)$) into the fraction acknowledged ($p(\cdot|\cdot)_a$) and the fraction concealed ($p(\cdot|\cdot)_c$). Although developers in the error-generating phase are aware of the need to coordinate and change concealed rework requirements with other phases, they do not alert other developers and initiate the required coordination. Therefore no action can be taken to address these changes. Based on this reasoning we model concealed change requirements in the same manner as tasks requiring rework that are not discovered during quality assurance, i.e. by approving and releasing the work. The acknowledged fraction of tasks undergoing quality assurance moves directly from the stock of work requiring checking to the stock of work requiring coordination through the discovery of rework flow (d). The concealed fraction is added to the flow of approved work (a).

$$p(\cdot|\cdot)_a = p(\cdot|\cdot) - p(\cdot|\cdot)_c \quad (17)$$

$$d = q * p(\cdot|\cdot)_a * p(\cdot) \quad (7a)$$

where: $p(\cdot|\cdot)_a$ - fraction of required rework discovered and acknowledged

$p(\cdot|\cdot)_c$ - fraction of required rework discovered but concealed

Notice that the total probability of a task requiring rework ($p(\cdot)$) and scope (S) are not changed, indicating a constant product complexity and project size. In calibrating our model to the Python project (Figure 5) we assumed that the pressure to show progress caused developers and managers of the Python project to conceal half of the rework requirements which they discovered except in the final quality control phase where all remaining rework is found and acknowledged. For simplicity we assume the fraction of discovered problems concealed is the same for internally generated rework needs and rework needs caused by inheriting flaws from supplying phases. However our explicit modeling of internally and externally generated rework allows the investigation of alternative assumptions such as

greater concealment of internally generated rework. Figure 7 shows simulations of the Python project with policies of concealing all and none of the rework requirements discovered. Notice that maximum concealment shows faster progress throughout the majority of the original 39 week schedule. This demonstrates that managers of individual development phases can temporarily use concealment to effectively address schedule pressure. However the concealment policy delays the last portion of the project into the characteristic pattern of the 90% syndrome, causing the project to be completed 87% (35 weeks) later than originally planned and 11 weeks (17%) later than the project managed without concealment. This illustrates that a concealment policy can cause or exacerbate the 90% syndrome. Concealment is locally rational (for individual phase managers) but globally irrational (for project management) and organizational success.

7. Discussion: Iteration Design and Management for Cycle Time Reduction

The frustration of competent and well-intentioned managers in concurrent development projects can be understood as resulting from process-constrained progress, magnified by concurrent development practices, and distorted by well-intentioned but narrowly focused management policies. Concealing rework requirements in response to the schedule pressure induced or exacerbated by concurrent development practices shifts the burden of and responsibility for change discovery away from individual concealing phases while improving apparent managerial performance. Since this creates no large deviations from the behavior of projects without concealment during most of the original project duration, managers do not receive signals that could initiate preventive action. By the time concealed changes are discovered and acknowledged, the project's structure has created a backlog of hidden rework. The resulting iteration cycles delay completion and increase cost even when project resources are ample because the process is constrained by the underlying recursive structure of information exchange within concurrent development. Since managers typically have less influence over processes than resources they have few effective tools and methods with which to accelerate throughput when iteration cycles constrain progress. Indeed, attempts to remedy these delays through overtime and hiring can worsen the problem through fatigue,

skill dilution, and other effects (Haddad 1996, Thomas and Raynar 1994). The process-behavior interaction helps explain how the 90% syndrome can occur in stable sized, competently managed, adequately staffed projects.

Effective strategies for reducing cycle time in concurrent development projects are not obvious. Our research suggests that an effective strategy addresses the managerial behaviors that cause iteration cycles to constrain progress as well as process drivers. Iteration cycles can delay projects by being more in number, longer in the distance which information must travel, slower in traversing that distance, and occurring later than possible. Researchers have proposed process designs to manage iteration cycle number, speed, length or timing. For example Terwiesch, Loch and DeMeyer (1998) recommend "a fast process of problem detection, problem solving and engineering change implementation" which increases iteration cycle speed. They suggest "loosening the coupling (dependence) between development activities" and improving the accuracy of preliminary information, both which reduce the number of cycles. McAllister and Backhouse (1996) suggest redesigning work flows to reduce the number of iteration paths in a project network. However all of these processes become more difficult to manage with increased concurrence. Process changes cannot improve concurrent development project performance if they do not also address the behaviors that drive iteration cycles such as the policy of concealing change requirements. Advanced information technology tools can play a role by acting on both processes and behaviors simultaneously. For example Sabbagh (1995) reports Boeing's use of computer assisted design and drafting (CADD) to identify conflicts among functional systems for use of the limited space in the aircraft. This reduced the potential for concealing change requirements as well as accelerating error detection and was an important factor in the success of the development project. Established technologies can also effectively address both information processes and development behavior simultaneously. Haddad (1996) describes how the innovative architectural design of an automobile firm's technical center for product development shortened communication paths (process) and increased the focus and commitment of development team members (behavior), thereby contributing to a cycle time

reduction of more than twelve months. Designing process and policy changes to fulfill both information processing and managerial behavior objectives can reduce cycle time more than addressing either in isolation.

In this paper we have used a dynamic model of a development project to describe, quantify and simulate how physical and information processes interact with managerial decision making to constrain progress and cause project overruns. We have shown the critical role of iteration cycles in explaining the 90% syndrome and how concealing discovered rework requirements is locally rational but globally irrational. Such concealment is common in development projects. It arises from the mental models of the managers and developers of individual phases and is amplified by local incentives and organizational cultures favoring concealment. Incorporating the interactions of development processes and behavior in development project changes can be critical for improving performance. This requires that practitioners understand the interactive roles of development information structures and participant behavior in limiting performance. Our model can illustrate these principles to practitioners and be used to design and test project management structures and policies, thereby improving project design and management.

Our work has several implications for concurrent development research. The model and simulations demonstrate that effective modeling of development processes must integrate the structure of information dependencies with the behavioral decision making of the actors. The role of iteration cycles in the 90% syndrome demonstrates the need for explicitly including iteration in project models. Tools and methods that reduce the opportunity for concealment of known rework requirements can play an important role in cycle time reduction. Future research can identify and test iteration cycle metrics that relate iteration to different forms of project and phase performance and how specific iteration features constrain concurrent development progress. Additional investigations of the interaction of information processes and managerial behavior can identify and describe other important linkages that constrain performance and design process and policy changes for improvement. Understanding how

development processes and managerial behavior interact can lead to policies to improve concurrent development projects.

References

- Abdel-Hamid, T. 1988. Understanding the "90% Syndrome" in Software Project Management: A Simulation-Based Case Study. *Journal of Systems and Software*. **8** 319-330.
- Backhouse, C. J. and N. J. Brookes 1996. *Concurrent Engineering, What's Working Where*. The Design Council. Gower. Brookfield, VT.
- Cooper, K. G. 1980. Naval Ship Production: A Claim Settled and a Framework Built. *Interfaces* **10**(6) 20-36.
- Ford, D. N. 1995. The Dynamics of Project Management: An Investigation of the Impacts of Project Process and Coordination on Performance. doctoral thesis. MIT. Cambridge, MA. .
- Ford, D. N., A. Hou and D. Seville 1993. An Exploration of Systems Product Development at Gadget Inc. Report D-4460. Sloan School of Management. MIT. Cambridge, MA.
- Ford, D. N. and J. D. Sterman 1998a. Dynamic Modeling of Product Development Processes. *System Dynamics Review*. **14**(1) 31-68.
- Ford, D. N., and J. D. Sterman 1998b. Expert Knowledge Elicitation to Improve Formal and Mental Models. *System Dynamics Review*. **14**(4) 309-340.
- Haddad, C. J. 1996. Operationalizing the Concept of Concurrent Engineering: A Case Study from the US Auto Industry. *IEEE Transactions on Engineering Management*. **43**(2) 124-132.
- Hauptman, O. and K.K. Hirji 1996 The Influence of Process Concurrency on Project Outcomes in Product Development: An Empirical Study of Cross-Functional Teams. *IEEE Transactions on Engineering Management*. **43**(2) 153-178.
- Krishnan, V. 1996. Managing the Simultaneous Execution of Coupled Phases in Concurrent Product Development. *IEEE Transactions on Engineering Management*. **43**(2) 210-217.
- Krishnan, V., S. D. Eppinger, D. E. Whitney. 1995. A Model-Based Framework to Overlap Product Development Activities. *Management Science*. **43** 437-451.

McAllister, J. and C. Backhouse. 1996. An Evolving Product Introduction Process. in Backhouse, C. and Brookes, N. (Eds.) *Concurrent Engineering, What Works Where*. The Design Council. Gower. Brookfield, VT. .

Meyer, C. 1993. *Fast Cycle Time, How to Align Purpose, Strategy, and Structure for Speed*. The Free Press. New York. .

Noreen, E., D. Smith and J. Mackey 1995. *The Theory of Constraints and its Implications for Management Accounting*. North River Press. Great Barrington, MA.

Repenning, N. P. and J. D. Sterman. forthcoming. Getting Quality the Old-Fashioned Way: Self-Confirming Attributions in the Dynamics of Process Improvement. In R. Scott and R. Cole (Eds.) *The Quality Movement and Organizational Theory*. Newbury Park, CA. Sage.

Rosenthal, S. R. 1992. *Effective Product Design and Development*. Business One Irwin, Homewood, IL.

Roth, G. and A. Kleiner. 1996. The Learning Initiative at the AutoCo Epsilon Program, 1991 - 1994. Sloan School of Management. MIT. Cambridge, MA.

Sabbagh, K. 1995. *21st Century Jet, The Making of the Boeing 777*. Pan Books. London.

Safoutin, M. J. and Smith, R. P. 1998. Classification of Iteration in Engineering Design Processes. *Tenth International ASME Design Theory and Methodology Conference*, Atlanta.

Smith, R. P. and S. D. Eppinger. 1997a. A Predictive Model of Sequential Iteration in Engineering Design. *Management Science*. **43**(8) 1104-1120..

Smith, R. P. and S. D. Eppinger. 1997b. Identifying Controlling Features of Engineering Design iteration. *Management Science*. **43**(3) 276-293.

Smith, R. and J. Morrow. Product Development Process Modeling. *Design Studies*. forthcoming.

Terwiesch, C. C.H. Loch and A. DeMeyer, 1998. A Framework for Exchanging Preliminary Information in Concurrent Development Processes. Working Paper 98/53/TM. INSEAD. Fontainebleau, France. 1998.

Thomas, H. R. and K. A. Raynar 1994. Effects of Scheduled Overtime on Labor Productivity: Quantitative Analysis. SD-98. Construction Industry Institute. Austin, TX.

Wheelwright, S. C. and K. B. Clark. 1992. *Revolutionizing Product Development, Quantum Leaps in Speed, Efficiency, and Quality*. The Free Press. New York. .

Womack, J. P., D. Jones and D. Roos, 1990. *The Machine that Changed the World, The Story of Lean Production*. Rawson Associates. New York.

¹ Associate Professor, Department of Information Sciences, University of Bergen, N-5020 Bergen, Norway <David.Ford@ifi.uib.no>

² J. Spencer Standish Professor of Management, Sloan School of Management, Massachusetts Institute of Technology, 50 Memorial Drive, E53-351, Cambridge, MA 02142 USA <jsterman@mit.edu>

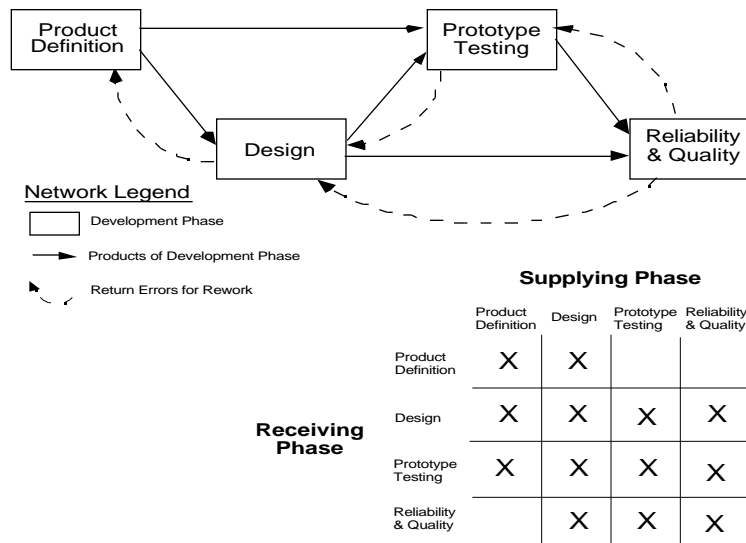


Figure 1: A Project Phase Network and its corresponding Design Structure Matrix

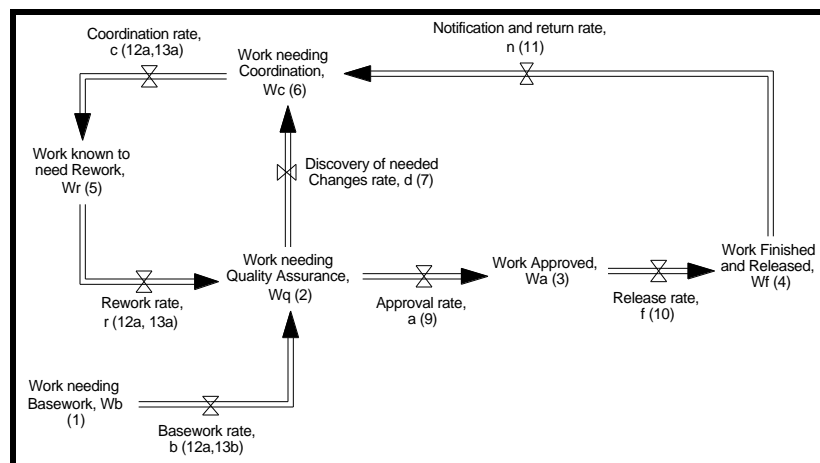


Figure 2: The Storage and Flow of Work in a Single Development Phase

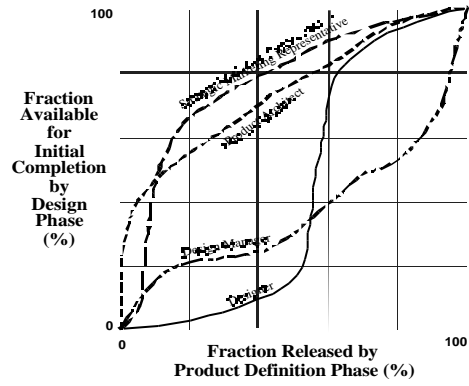


Figure 3: Four Estimates of the External Concurrence Relationship between the Product Definition and Design Phases

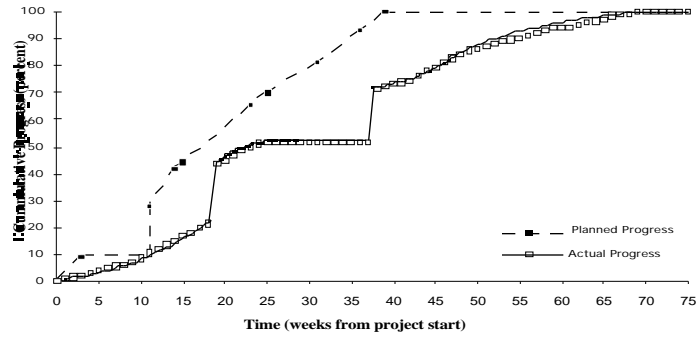


Figure 4: Planned and Actual Progress of the Python Project

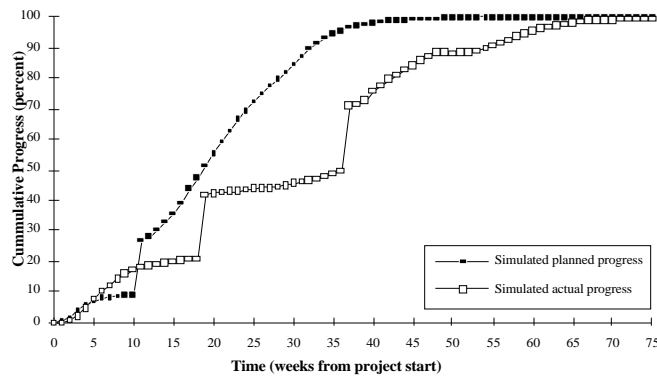


Figure 5: Simulated Planned and Actual Progress of the Python Project

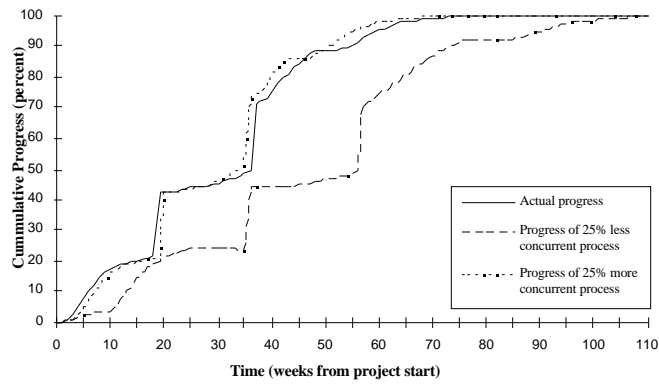


Figure 6: Python Project Simulations with different levels of Concurrency

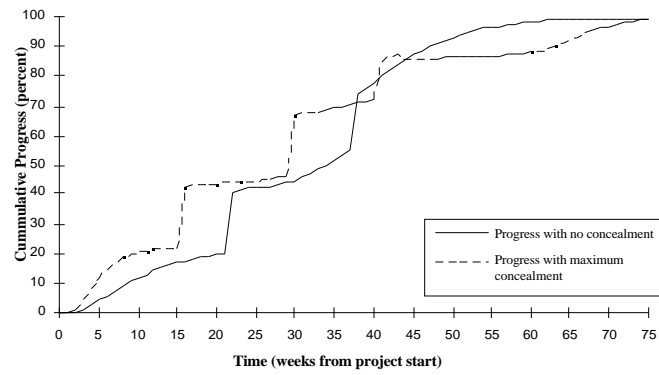


Figure 7: Python Project Simulations with different levels of Rework Requirement Concealment