Int J Software Informatics, Volume 5, Issue 3, 2011, pp. 443-456
International Journal of Software and Informatics, ISSN 1673-7288

E-mail: ijsi@iscas.ac.cn
http://www.ijsi.org
Tel: +86-10-62661040

Organizing projects around the mitigation of risks arising from system dynamic behavior

Neil G. Siegel*

(Northrop Grumman Corporation, United States)

Abstract Many of the key products and services used by modern societies are the result of large-scale engineering projects. Despite decades of theoretical and practical work in the art of systems engineering and project management, project execution results remain somewhat inconsistent, in the sense that many projects fail to produce a product that meets the original specifications, and many more projects achieve some measure of technical success only after taking significantly more time and/or money than originally expected. One source of such failures is the occurrence of unplanned and adverse dynamic behavior in the resulting system. This paper summarizes research being conducted to look at the potential of design-phase actions that centralize control of the eventual system's dynamic behavior as a potential solution to some instances of this problem. This approach could lead to increased chances of success on future major system development projects, through a new method for instituting better control of the dynamic behavior of such a system.

Key words: dynamic behavior of systems, large-scale systems, system development projects, system architecture skeleton, SAS, defect rates, error rates.

1 Introduction

Many of the key products and services used by modern societies are the result of large-scale engineering projects. Despite decades of theoretical and practical work in the art of systems engineering and project management, project execution results remain somewhat inconsistent, in the sense that many projects fail to produce a product that meets the original specifications, and many more projects achieve some measure of technical success only after taking significantly more time and/or money than originally expected.

One source of such failures is the occurrence of unplanned and adverse dynamic behavior in the resulting system. This paper summarizes research being conducted to examine the hypothesis that use, during the design phase of a large system development project, of a design-based technique that centralizes the control of system dynamic behavior could improve system development outcomes, by lowering the density of those defects that are attributable to unplanned adverse dynamic system behavior. These improved outcomes would manifest themselves through fewer system development failures, schedule delays, and cost over-runs.

I have developed such a technique to address this issue, in the context of a specific class of systems, within a specific application problem domain in which I have industrial experience. Through this industrial experience, I have applied this technique to multiple large projects over a period of nearly 20 years, with some apparent success. This paper will describe this problem, describe the technique I developed and deployed to correct it, use the data from a number of these completed real-world project applications to assess the efficacy of the technique, and discuss its improvement,

* Corresponding author: Neil G. Siegel, Email: Neil.Siegel@ngc.com
Received 2011-02-09; revised 2011-02-17; accepted 2011-02-26; published online 2011-03-11.

application, and potential extension to additional problem domains.

2. Summary of the approach

The study whose results are summarized herein takes the form of an observational case study. Drawing on data drawn from four real-world projects (the primary case, and three secondary cases), the case documentation has been collected and organized. The independent and dependent variables, the measurement instruments, and the processing protocols are all defined. The measurement instruments are then applied to the cases, resulting in measurements and statistics. Finally, an interpretation of what the cases indicate about the hypothesis is formulated.

3. Scope of the systems-of-interest

The use of complex systems seems to be on the increase, per sources such as (Ramo & Booton 1984)[1] and (Boehm 2010)[2]. These systems perform important roles for society, running our power grid, our banking system, our air-traffic control system, our traffic signals, and so forth.

Herein, however, I am interested only in systems with certain specific characteristics:

- Complex emergent behavior, as described by (Rechtin 1991)[3]
- Interactions with physical devices (physically-moving mechanisms, other time-sensitive mechanical devices, etc.)
- Stressing asynchronous stimuli (such as extraordinary high data-ingest rates, or highly-stressed communications structures)
- Extraordinarily high availability / reliability requirements
- Development efforts of large size
- Systems that need to display much early progress through prototyping and re-use, but need, at the same time, to avoid having their design "locked in" to a pattern that will be ineffective over the life of the development effort

Within this study, I will term systems that display these characteristics "large-scale complex systems"; this study is limited to systems with these characteristics. As will be seen, such systems are still "interesting", in the sense that there are a lot of such systems, these systems experience development-phase failures, and this subset still spans a meaningful range of applications problem domains.

Within such large-scale complex systems, there is a material possibility of a high degree of "fan-out" of options along the system processing threads, which can create system state-spaces that are larger than might be traversed by a nominal test activity; this increases the likelihood of the system inadvertently progressing into a state where some aspect of system performance becomes unacceptable. A typical behavior experienced when a system thus progresses into an unintended state is the creation of race conditions, or other unexpected sequencing, resulting in various sorts of errors, including the cessation of meaningful operations. This of course lowers system availability. Systems where asynchronous stimuli are significant, either through exceedingly high stimulation rates, or through communications-media-usage being near the theoretical limit (which tends to introduce non-linear degradations akin to turbulence in water flow), are particularly vulnerable to this type of problem.

In this study, I propose a candidate corrective method, and measure its efficacy. There is a cost associated with implementing this method, and therefore, I limit my

assessment to larger systems (e.g., total non-recurring development effort greater than 1,000 man-years); projects that are materially smaller may have not have the technical and social complexity that would justify the investment in the cost of implementing the design-based technique described herein.

The element of social complexity induced into large projects is worth special mention: most projects of this size have far more constraints placed on them than smaller projects (social complexity seems to increase far more than linearly with project budget size). These social constraints may take the form of requirements to use a specific off-the-shelf product or products as a part of the implementation, to co-exist with legacy system and their interfaces for large portions of their operational life-time, constraints about specific contractors / vendors to use (including geographic goals), and so forth. These social constraints tend to close off portions of the design trade space, and often leave projects with designs that have that large "fan-out" of potential system states described above. Techniques like that considered in this study can help "buy-back" system performance lost to such constraints.

In addition, projects of this size often have an imperative to show progress quickly (often, even during the pre-award time-period), through prototyping and re-use; this leads (perhaps inadvertently) to an emphasis at the beginning of the development effort on implementing the "easy" and/or most visible portions of the problem (either those that are inherently easy, those portions of the system for which partial solutions are available through re-use, or those portions of the system that provide the user interfaces). Focusing on these portions of the problem does indeed allow for a lot of apparent progress to be demonstrated, and therefore, to meet the show-progress objective mentioned above. But so placing initial focus on these "easy" and visible portions of the system may not be a path to a viable total system – because focusing on the easy / re-use-based / visible portions of the system may invoke early design decisions that might inadvertently preclude achieving important system goals as the total system progresses. For example, an early decision to re-use off-the-shelf or otherwise existing components can lead to high external complexity at the interfaces[4], which can in turn lead to the problems noted above.

This is particularly the case for behavior that appears only as the system nears completion, that appears only as the induced load approaches its design limits, or that appears only as the system gets used over long periods of time. The design-based technique discussed herein is in some sense a "hard-part-first" strategy, and is intended explicitly to address this issue. Much of the literature regarding early prototyping, however, has in fact focused on system functions like the user interfaces, which are visible, but technically, relatively unchallenging. Yet, given data from sources like (Forman & Cureton 2010)[5] about the likelihood of large, long programs being cancelled before completion (their thesis is that the long development schedules associated with large programs create temptations for competitors to arise, and when significant technical problems arise mid-project, especially those that seem fundamental to the selected design, programs are cancelled either to try one of the competing approaches, or to allocate the funding to another mission), a more balanced approach that allows "front-end" demonstrations of quick progress combined with efforts explicitly intended to lower the risk of material problems arising mid-program (e.g., "hard part first") seems worth investigating.

The characteristics described above form the basis for defining the types of systems of interest to this study. My professional experience includes systems for the U.S.

Government, for the entertainment industry, for real-time process control in manufacturing settings, electronic medical records, Enterprise Resource Planning, enterprise information, logistics automation, radar and other sensors, airplanes, robots, and others. This experience suggests that the findings of this study will apply across all of these problem domains, e.g., there will be some material number of systems in each of these application problem domains that incorporate the characteristics listed above. Interactions with colleagues, participation on industry and U.S. Government senior advisory panels, and readings in the literature lead me to believe in the likelihood that these findings will be applicable in additional domains with which I do not have personal experience.

4 The design-based technique

A modern large-scale, complex, software-based system, perhaps with hundreds (or even, thousands) of computer processors operating in a distributed fashion, each of which has hundreds of independently-scheduled software entities, and many forms of asynchronous stimulation (including from people), can have many opportunities for such unplanned / adverse dynamic behavior. The adverse effect can be highly non-linear – that is, seemingly small instances of unplanned dynamic behavior can seriously degrade and compromise a seemingly well-designed and well-tested system.

The introduction of ever-larger amounts of software into large-scale systems [described by (Boehm 2010)] introduces an additional significant increase in the potential interconnectedness. Not only are there many more "parts" – the systems under consideration herein have a few tens of thousands of physical parts, but millions of lines of software – but unlike hardware components which only directly interact with other parts with whom they are physically interconnected (via a wire or near-field electromagnetic effects, for electronic parts; via mechanical motion, for physical parts like gears and levers), software "parts" can directly interact (via various sorts of calls, etc.) with almost any other software "part". So, in software there are both many more "parts", and the potential for a much denser interconnection topology. This has led to a situation where unplanned adverse dynamic behavior is a significant factor in the failure mechanisms of large-scale, software-intensive system development. (Rechtin 1991) points out the central role of emergent behavior in a system, much of which is associated with dynamics; (Siegel 1993)[6] points out that not all of this emergent behavior is desirable, and that explicitly defining the desirable emergent behavior, and then structuring the control logic of the system to be support that behavior and no other (which might be undesirable emergent / dynamic behavior) is a key systems engineering role.

Modern software practices have (unintentionally) tended to increase the likelihood of such problems; for example, the increasing use of service-oriented structures in software systems, in which hundreds of software entities are made available to be called and used by a large set of applications, has led to a significant increase in such unplanned dynamic behavior – the developers of these software services may have no credible way to understand all of the ways in which their service may be used.

Yet there is little perceivable momentum towards society wanting to give up the functionality and richness achievable through such large software-intensive systems – these perform vital roles for society that cannot, at present, be performed in any other manner. It would, therefore, be desirable to address and correct the problem, so that

society can realize the benefits of such systems, yet the development process for such systems will succeed more often.

Dynamic behavior is implemented in such large-scale systems through a chain of events such as the following:

- An asynchronous stimulation (e.g., from a person, from a communication channel, from a sensor, etc.), or a synchronous stimulation (e.g., from a timer) occurs, forming an input to the system. This can be mechanical or electrical.
- This stimulation causes the spawning of a software or hardware action, to respond and process the event or input.
- This can cause a thread (planned sequence of activity). Note that any step along the thread, however, may spawn additional, concurrent threads.
- Eventually all of the steps of the original thread complete, and these software / hardware components then remain quiescent until the next instance of stimulation occurs.

There are at least two classes of problems that might occur in the context of such a thread execution:

- Many of these threads have hundreds or even thousands of potential branches and alternatives, which often include the spawning of separate and concurrent additional threads. So even within the context of a single thread initiation, unplanned dynamic behavior can occur, through the interaction between the spawned threads.
- But the more common and more serious instances generally arise because the stimuli for the above threads are often intrinsically asynchronous, and an indeterminate number of combinations of such threads may be initiated and running concurrently.

The complexity associated with analyzing the above is far beyond the capacity of the tools generally available; for example, (Medvidovic 2010)[7] points out the large combinatorial complexity of a similar analysis problem, and in general, those instances that prove the most troublesome are precisely those instances of dynamic system behavior that were not foreseen (e.g., were unplanned). Some [such as (Dijkstra 1998)[8]] call for solving this by programming only through the artifice of formally-proven mathematical models; even, however, if desirable, this is not a typical current practice, and there are no signs of its being adopted on a large scale. Others, such as (Madni 2008)[9] call for attempting to address this sort of problem through what he terms "resilience engineering".

This problem is made more difficult by the fact that the typical software environment has many mechanisms for implementing concurrent processing. For example, usually the operating system has more than one such mechanism, usually each layer of the system's middleware provides at least one, and most modern programming languages have one or more such mechanisms built right into their language definition.

Furthermore, the typical system development methodology distributes the decision-making about the implementation and control of concurrency (and other forms of dynamic behavior) across the entire population of developers; every software developer (and many of the hardware designers) can trigger additional concurrent processing threads within the system.

The process fails, therefore, because:

- Having the responsibility for the implementation of dynamic behavior distributed across the members of the development team allows the potential for

unplanned dynamic behavior to be introduced by design decisions – the individual designers may not understand all of the ways in which their component can participate in processing, and may therefore not provide adequate controls to ensure that their component performs always as planned.

- Such distributed responsibility for the implementation of dynamic behavior makes an implicit decision that any member of the development team is adequately skilled in the area of controlling dynamic behavior to be able to make appropriate design decisions. Given the wide range of skills and experience on a large team (the example projects considered in this study each had on the order of 300 to 500 developers), this is likely an incorrect assumption.

Figure 1 provides a procedural description of the design-based system architecture skeleton (SAS) technique.

Figure 1 starts with the step of creating a description of the system's desired dynamic behavior, down to the level of every independently-schedulable entity and event within the system. These independently-schedulable entities – hardware and software – form the building-blocks from which system dynamic behavior is formed. Any reasonable endeavor to control and improve system dynamic behavior must include a step of capturing a list of these building blocks.

The figure continues with a design decision – to centralize control of system dynamic behavior into a small portion of the design and implementation, and one that will therefore be implementable by a small, expert team.

The process continues with a cataloging of all of the forms by which processing initiation and especially, concurrency, can be introduced into a system; e.g., each of the levels of software (operating system, programming languages, middleware scripts, etc.), hardware interrupts, true simultaneous external stimulation, and so forth. One then selects an appropriate subset of these mechanisms as being the only ones that will be available to the development team, e.g., we will use the provisions of the middleware for thread dispatch and rendezvous, but not use the similar features built into the operating system and programming languages. One must then train the developers regarding these decisions, and create mechanisms (such as code auditors and peer-review procedures) for enforcing those proscriptions.

Having catalogued both the independently-schedulable entities and the forms for introducing stimulation and concurrency, one can then form processing thread definitions that can be carried through to the level of capturing the desired invocation conditions for every independently-scheduleable entity within the system. For most real systems, this is complicated enough that some sort of computer-readable representation of these data is probably required, allowing this representation to become an actual specification of the desired processing threads and sequences, and most importantly, to become a "script" that is used to create a control structure for the system.

One must then provide an executable mechanism that can read this computer-readable representation of processing threads, and turn it into processing instructions, using only the selected mechanisms for initiating processing and concurrency. The intent is that (a) this control kernel is a small portion of the overall system implementation, and (b) is the only mechanism that can invoke processing or concurrency. It must literally block non-authorized invocations. It takes experience and some trial-and-error to reach a balance between achieving a small implementation with rigorous blocking of non-authorized behaviors, and sufficient richness of

expression to support the range of processing required in a real-world system. In the case described in this study, we did not reach a satisfactory balance until the 3rd iteration.

Having achieved all of the above in advance of taking the (presumably large) effort to develop all of the systems actual applications software and specialized hardware devices, one can still implement a system architecture skeleton that implements the actual processing threads – using the actual intended control mechanisms, both hardware and software – in a manner that can have credible timing, capacity, and other characteristics that will reflect the end-state of the eventual system. Elements not yet available can be represented by stubs that reflect their predicted resource utilization (e.g., port-to-port timing, communications utilization, and so forth). In the case described in this study, this involved only about 1% of the total system development effort, so this step credibly could preceed most of the development effort.

The key step (shown in boldface type in Figure 1) is the step that accomplishes the actual de-coupling of the "hard" work from the more normal work – allowing for the entities within the system that are designed and implemented by those outside of the small team implementing the control kernel to implement their elements of the system without reference to the desired system-level dynamic behavior, yet for their components to in some fashion inherit a set of behavioral controls in accordance with the parameters and algorithms in the control kernel. One approach to achieving such inheritance is the use of pre-written shells that control processing invocation and concurrency for software applications. Other approaches have proven feasible, as well.

With the additional of instrumentation and stubs, the threads can be exercised, and system performance measured, even in the absence of actual hardware devices and target application software through the use of the system architecture skeleton, which incorporates the actual implementation of the control structure from its initial instantiation.

The design-based technique assessed herein to correct this problem consists of both a *methodology*, and a set of *implementing / supporting tools*.

The design-based technique assessed herein to correct this problem consists of both a *methodology*, and a set of *implementing / supporting tools*. These are described in Figures 2 and 3.

The methodology portion of the design-based technique must provide the following capabilities:

- Allow for the description and documentation of system dynamic behavior, which implies that this specification must reach to the level of every independently-schedulable entity within the system; it is telling that relatively few system- or software-development guidance documents [e.g., (INCOSE 2007)[10], (Humphrey 1995)[11], etc.] even address the concept of independently-schedulable entities.
- Allow for the recognition of every mechanism available to a developer that could create concurrency and other forms of dynamic behavior within the system, and train the developers outside of those implementing the control kernel that they are not to use these mechanisms.
- Enforce that proscription.
- Allow for the specification of the threads and allowable interconnects / interactions amongst the components (whether hardware or software or people).

- Allow for the instrumented exercising and analysis of the system threads under realistic stimuli, and do so in a way that allows the dynamic behavior to be observed and adjusted separately from the specific functionality of the system.

The implementing / supporting tools must:

- Allow for a small portion of the implementation to specify and control the dynamic behavior of the system, e.g., provide a sort of "control kernel".
- Allow for the entities within the system that are designed and implemented without reference to dynamic behavior in some fashion to "inherit" a set of behavioral controls in accordance with the parameters and algorithms implemented within the control kernel.
- Automatically translate the specification of the threads and allowable interconnects / interactions amongst the components (whether hardware or software or people) into an executable mechanism (the scale and complexity is usually far too large to do this credibly and reliably by hand).
- Provide an execution and instrumentation framework for the dynamic system threads, even prior to the availability of actual system components (hardware and software).

In this design-based technique, the boundary between systems engineering and software development is blurred; this is consistent with the findings of (Boehm 2010), who cites efforts and recommendations for more integration between the processes and approaches of the two fields.

5 Results

As noted above, the hypothesis is "During the development phase of a large-scale, complex computer-based system, the use of a design-based technique that centralizes the control of the dynamic behavior of a system will lower the density of those defects that are attributable to unplanned adverse dynamic system behavior". This hypothesis has been assessed against multiple cases, described below.

I start the assessment with project YYYY, the primary case. This project has had about a dozen major system test events over its 15-year life; some of these are in what I term "period I" (initial use of the technique); some are in "period II" (non-use of the technique); some are in "period III" (return to the use of the technique). I have selected the life-cycle-point of entry into and performance of the contractor-conducted system-level formal acceptance test as a place to collect comparable defect data from the three periods; there have been such test events in each of the three periods. I have gone through the actual error report logs from the relevant portions of periods I, II, and III, and identified errors that I believe are directly attributable to errors in the control of system dynamic behavior, and created monthly totals. I have performed this analysis separately for each of the three periods, and combined the results with the 3 secondary cases into Figure 4, below. The y-axis unit is density of attributable error reports opened per month, on a scale where 1 = 1 report per 1,000,000 source lines of code.

In looking at Figure 4, what the data appears to indicate for project YYYY is that during period I (when the project was using the design-based technique to centralize the control of system dynamic behavior) the project had a consistent new-occurrence rate of defects due to errors in controlling system dynamic behavior on the order of 3 per month. During period II (when the project was *not* using the design-based technique

to centralize the control of system dynamic behavior), the "voice of the process" is quite different – the average new-occurrence rate of defects due to errors in controlling system dynamic behavior is around 18 per month, and displays a larger range of variation than the data for period I and period III. The rate during period III (when the project was *again* using the design-based technique to centralize the control of system dynamic behavior) was similar to period I (e.g., around 3 per month).

The next question is whether or not this change in the "voice of the process" can appropriately be attributed to the change in the independent variable (e.g., the use or non-use of the design-based technique). This was analyzed through a "plausible rival hypothesis" methodology; seven plausible rival hypotheses were generated and assessed. None of them in fact turned out to be plausible, and I therefore concluded that attributing the observed change in the dependent variable to the change in the independent variable was justified.

I have also done some assessment of the cost-performance data that corresponds to the periods I through III. The contract value to-date is about $2.5B US ($2,500,000,000 US). Although all awarded under the aegis of the original competitive award, a large number of separate contractual delivery orders were used, facilitating separation of cost and schedule against budget for individual activities. In general, development, production, and services (e.g., training, maintenance, etc.) were contracted on separate delivery orders. Individual versions of development capability were generally contracted on separate delivery orders, as well. This provides far more insight that would be available looking at aggregate cost / schedule performance for the entire contract.

For example, the period-I development efforts that lead up to the capability tested in 1998 (e.g., the development effort that took place from 1995 to 1998) show a modest cost under-run (1% over-run to 4% under-run on the various delivery orders). The period-II development efforts show significant and consistent over-runs (15% to 25%). The period-III developments show a return to the modest under-run pattern of period-1. I have excluded costs that are not relevant to non-recurring development from the above, e.g., hardware production, and post-delivery services (e.g., installation, training, maintenance, etc.). That is, the cost performance on the project seemed to vary in synchronicity with the dependent variable of the study.

Next, I consider the secondary cases. First is Project ZZZZ, which I assessed during a 2002 test cycle. Since 1989, Project ZZZZ has been using the design-based technique. The next secondary case is project AAAA; this project did not use the design-based technique. The final secondary case is project BBBB; this project did not use the design-based technique.

Figure 4 incorporates the data for projects YYYY, ZZZZ, AAAA, and BBBB. A marked grouping appears in Figure 4 – three data sets cluster around low values (e.g., 0 to 4 new attributable reports per month), and three data sets cluster around higher values (e.g., 15 to 27 new attributable reports per month). The three data sets with the low values correspond to those instances where the design-based technique was used (Project YYYY period I, Project YYYY period III, and Project ZZZZ). The three data sets with the higher values correspond to those instances where the design-based technique was not used (Project YYYY period II, project AAAA, project BBBB).

6 Conclusions

The data presented indicate that the proposed design-based technique for centralizing the control of dynamic system behavior during the design and implementation phase of a complex system development may in fact lead to better outcomes. In this instance, this was measured by the materially-lower density of defects that were attributable to errors in controlling such system behavior in the projects and periods that used the design-based technique than for those that did not use it. Additional measures are possible.

It also turned out that such errors formed a large percentage of the most serious errors found in the systems examined for this study. Each of the four systems studied classified all reported errors into a ranking scheme, in terms of the impact of the error upon the system. For example, in that period of Project YYYY where the design-based technique *was not* used (period II), the errors attributable to the cause under investigation accounted for 92% of the total number of serious errors reported. Since the number of attributable errors was much smaller during period I and period III (when the design-based technique *was* used), and there is no apparent increase during these periods in other causes of significant errors, the use of the design-based technique is apparently leading to a significant reduction in the *total* number of serious errors.

As noted above, for Project YYYY, cost performance (a key measure of customer satisfaction) tracked the behavior of the defect rate – that is, better when the design-based technique was used, and worse when the design-based technique was not used.

Although partly subjective, available measures of customer satisfaction with the outcome of the project also closely tracked the use / non-use of the design-based technique. That is, data collected from customers (including objective data, such as award-fee scores; and subjective data, such as interviews with customer personnel) indicate that customers appeared to be happier with the outcome of the development project in those instances where the design-based technique was used, and less pleased with the outcome when the design-based technique was not used.

If repeatable, and especially, if extendable into other application problem domains (all four cases reported herein were in the same application problem domain), this is a promising result. Given data like that from (Glass 2001)[12] which indicate that only 16% of such large-scale system development project are considered successes even by their own developers, the consistent pattern of success across the cases that used the design-based technique is striking.

I continue with studies to extend the scope of this study into four additional application problem domains (logistics automation, air-traffic control, radar, and large-scale information processing systems).

About the Author

Neil Siegel is Vice-President & Chief Engineer of the 25,000-person Information Systems sector within the Northrop Grumman Corporation. Among many other honors, he is a member of the United States National Academy of Engineering[13], a fellow of the IEEE[14], a member of the Order of Saint Barbara[15], and the 2011 winner of the IEEE Simon Ramo Medal[16] for systems engineering and systems science.

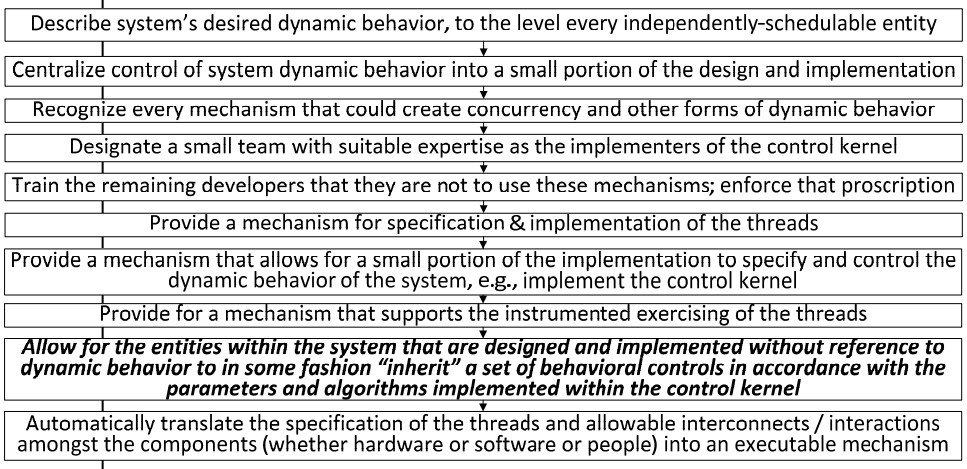

Figure 1. Procedural description of the system architectural skeleton.

Required features: Methodology	Specific element of the SAS approach that provides this feature
Allow for the description and documentation of system dynamic behavior (down to the level of every independently-schedulable entity within the system)	A specialized representation has been created, with appropriate semantics. The representation can be created manually (e.g., using a tool like Visio), but automation support has also be created.
Allow for the recognition of every mechanism available to a developer that could create concurrency and other forms of dynamic behavior within the system, and train the developers outside of those implementing the control kernel that they are not to use these mechanisms	Auditing checklists have been created that provide insight about how to prepare a comprehensive version of such a list for a specific project and its environments.
Enforce that proscription	A combination of (a) guidance for project reviews (e.g., reviewing this matter in peer reviews and code audits, with guidance provided about topics and questions), and automated means (e.g., code auditors, disabling operating system and compiler features, etc.).
Allow for the specification of the threads and allowable interconnects / interactions amongst the components (whether hardware or software or people)	A specialized representation has been created, with appropriate semantics. The representation can be created manually (e.g., using a tool like Visio), but automation support has also be created.
Allow for the instrumented exercising and analysis of the system threads under realistic stimuli, and do so in a way that allows the dynamic behavior to be observed and adjusted separately from the specific functionality of the system.	A tool has been created for this purpose (see below), and a method for using that tool.

Figure 2. SAS – required features – methodology.

Required features: Implementing tools	Specific element of the SAS approach that provides this feature
Allow for a small portion of the implementation to specify and control the dynamic behavior of the system, e.g., sort of a "control kernel"	This is performed by a tool we call the "software backplane", and shell templates for system applications. In combination, these items can implement, control, or disable dynamic behavior of hardware, software, and human interactions. The specific behavior is defined through scripts, in a syntax that we defined. This scripting is similar to BPEL (business process execution language[1]), but more complete with regard to controlling dynamic behavior.
Allow for the entities within the system that are designed and implemented without reference to dynamic behavior to in some fashion "inherit" a set of behavioral controls in accordance with the parameters and algorithms implemented within the control kernel	The shell templates (we call them "system manager programmer interface") perform this function.
Automatically translate the specification of the threads and allowable interconnects / interactions amongst the components (whether hardware or software or people) into an executable mechanism (the scale and complexity is usually far too large to do this credibly and reliably by hand)	The system management scripts are automatically turned into executable commands and initialization parameters for the relevant operating system, compiler, loader, and attached hardware devices.
Provide an execution and instrumentation framework for the dynamic system threads, even prior to the availability of actual system components (hardware and software).	The execution environment created by the system management scripts is built up out of the shell templates. The contents of the shell templates can include real production software and hardware, but can as be prototypes, models, and stubs, in any combination. This allows the dynamic behavior of the system to be observed, instrumented, and corrected separately from the functionality of the system.

Figure 3. SAS – required features – implementing tools.

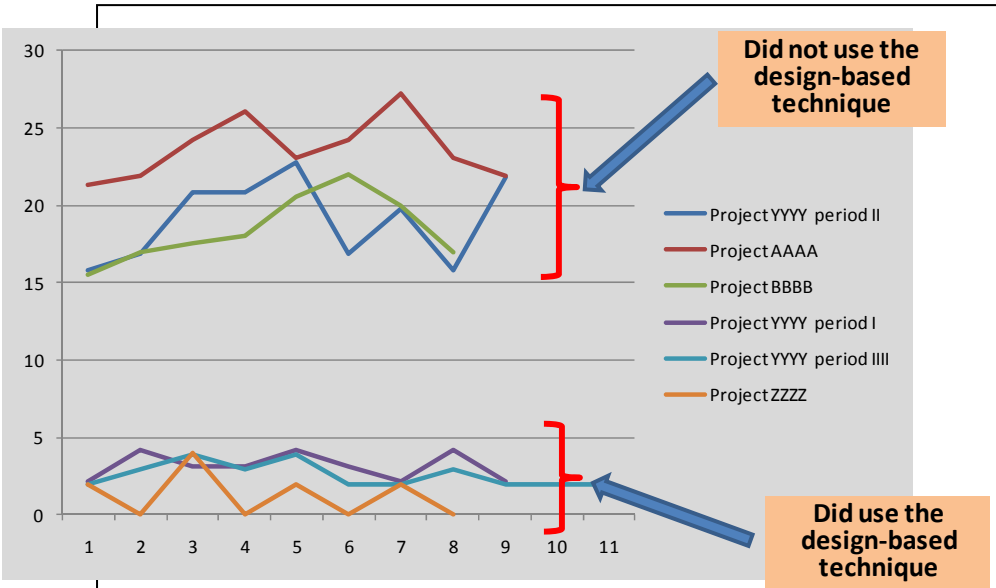

Figure 4. All six data sets on a single plot.

References

[1] Ramo, Simon & Booton, Richard, "The Development of Systems Engineering", IEEE Transactions on Aerospace and Electronics Systems, July 1984.

[2] Boehm, Barry W., "Some Future Software Engineering Opportunities and Challenges", pre-publication copy, 2010.

[3] Rechtin, Eberhardt; *Systems Architecting*, Prentice Hall, 1991.

[4] Rechtin's heuristic for good system design is for low external complexity (which includes low complexity at the interfaces), while permitting high internal complexity.

[5] Cureton, Kenneth, & Forman, Brenda, "Engineering Management of Government-Funded Programs", USC School of Engineering, 2010.

[6] Siegel, Neil, "Lessons-Learned with Ada in Building the Forward-Area Air Defense Command, Control, and Intelligence System", a chapter in *Ada: Lessons Learned in Development and Management*, TRW, February 1993.

[7] Medvidovic, Nenad, "Software Architecture and Mobility: A Perfect Marriage or an Uneasy Alliance?", lecture (with briefing charts) delivered 14 September 2010.

[8] Dijkstra, Edsger, "On the Cruelty of Really Teaching Computer Science", an open letter, 1988.

[9] Madni, Azad M., "*Towards a Conceptual Framework for Resilience Engineering*", Accepted for publication in the IEEE Systems Journal, Special Issue in Resilience Engineering, 2008.

[10] INCOSE, Systems Engineering Handbook, August 2007

[11] Humphrey, Watts, *A Discipline for Software Engineering*, Addison Wesley, 1995.

[12] Glass, Robert L., *ComputingFailure.com*, Prentice Hall, 2001

[13] http://nae.edu/

[14] Institute of Electrical and Electronics Engineers, http://ieee.org.

[15] See http://en.wikipedia.org/wiki/Order_of_Saint_Barbara.

[16] http://www.ieee.org/about/awards/medals/ramo.html.