The USC System Factory Project

Walt Scacchi
Computer Science Dept.
University of Southern California
Los Angeles, CA 90089-0782
(Scacchi@VAXA.ISI.EDU)

Abstract

Developing the next generation of large-scale software systems will change the patterns of work in system development organizations. We therefore expect the major systems engineering problems to be solved will require *organizational solutions* that accomodate *advanced software development tools, flexible manufacturing techniques for system life cycle engineering,* and *knowledge-intensive strategies for managing large system development projects.* Over the past seven years, we have created an experimental organizational environment for developing large software systems that allow us to encounter these problems, and find effective solutions or interventions. We call this organizational environment the *System Factory*. We have developed and evolved the System Factory through seven generations of graduate student staff, totaling more than 500 in number. In this report, we describe what the System Factory is, the problems we have investigated, the results and example products of this research, potential future applications of the System Factory approach, and finally our experiences in transferring this technology.

The System Factory Project has been supported over the years through contracts, grants, or gifts from the USC Faculty Research Innovation Fund, AT&T Information Systems, Bell Communications Research, Carnegie Group Inc., Eastman Kodak Inc., Hughes Radar Systems Group, IBM through Project Socrates at USC, System Development Foundation, and TRW Systems Engineering and Development Division. Additional support was provided by DARPA contract MDA 903-81-C-0331 to the Information Sciences Institute at USC. Finally, more that 500 graduate students in computer science at USC have elected to participate in the System Factory project since 1981. Without their participation and commitment to success, this project would not occur. We are truly grateful for all of their support

---

[1]Appears in *Proc. Software Symposium '88*, (Keynote Address), Software Engineers Association, Tokyo, Japan, pp. 9-41, (June 1988). Previously, it appeared as Technical Report CRI-87-67, Computer Science Dept., University of Southern California, Los Angeles, CA

1. Introduction

The System Factory project is concerned with exploring alternative ways and means for meeting the challenge of large-scale software system engineering (LSSE) in the 1990's. The System Factory approach consists of applying recent results in *software engineering, knowledge-based systems technology, computer-aided manufacturing*, and *organizational analysis of computing work* to the problems of engineering large-scale computing systems. In particular, these latter results outline many organizational problems in system development, as well as strategies for mitigating them that go beyond what existing engineering activities alone can accomplish.

Through seven years fo work in the System Factory project, we have produced and documented a variety of results and products. These outcomes include both technological and organizational artifacts, and numerous research contributions. We have produced an inventory of *reusable software components* that we can reconfigure into different application systems or environments [39, 20, 41] [31, 8, 44, 45]. Each of these components has a record of formal specifications and narrative descriptions that characterize its development life cycle. We have produced a set of *techniques for engineering the life cycle* of software applications and environments [32, 46]. These techniques primarily address how to articulate and transform software system specifications into concrete source code realizations, whether employing existing software components, or through prototyping entirely new application systems. We have produced and continue to refine a set of *strategies for managing LSSE projects* that can be specialized to specific organizational and technological arrangements [26, 51, 42] [43, 49, 3]. These strategies employ policies for project management that we have observed realize substantial improvement in software productivity and quality. We are also continuing our investigation to develop a *paradigm for flexible manufacture of large software systems* [12], and articulating a *knowledge base of software technology transfer know-how* [38, 48] based upon our research into effective transfer and transition practices.

Our simultaneous focus on system engineering activities, and the organizational patterns and processes in which they occur, is the unique aspect of our approach. It also represents, in our view, the best opportunity for realizing substantial improvements in software productivity, quality and long-term software cost reduction. We believe that our research results and products increasingly substantiate this.

Since the SF project exists within an academic setting, we have been able to work with large software development staffs of 30-70 MS and PhD graduate computer science students.[2] We started in 1981 with a basic time-shared computer system, a staff of 50+ students, and the initial SF concept. Over the last seven years, we have iteratively developed, used, and evolved the SF into its present form through seven development cycles. Thus, we also have produced more than 500 software engineers who have become skilled with contemporary tools, techniques, and management strategies useful in LSSE projects [47].

In this report, we first provide some background on the System Factory approach to LSSE. Section 2 serves to identify our view of some of the outstanding problems of LSSE. Section 3 next describes the emergence of the System Factory project over the past seven years. Section 4 describes the results emerging from the System Factory approach to LSSE over the last seven years. This includes a brief description of selected automated tools, development techniques, and setting-independent project management strategies we have produced and refined. In particular, we describe an ensemble of SF tool components that represent a reconfigurable computer-aided software engineering environment (CASEE) that operates on a network of hetereogeneous processors. In Section 5, we briefly suggest how the SF approach could be applied in the future to develop an automated environment for the design and evolution of a next-generation multiprocessor system. Section 6 describes our experiences in transferring System Factory technology locally and externally. Finally, we conclude in Section 7 with some reflections

---

[2]Available staff are typically group into 6-12 subcontracting teams consisting of 4-6 students. SF staff are expected to commit 10-12 hours per week to project activities, although they sometimes end up spending much more time. Because of our geographical setting, most SF students are actually software professionals from system development firms including AT&T, Hughes, TRW, Aerospace Corp., JPL, etc. working part-time toward a graduate degree in computer science. As such, some teams primarily work off-campus, since all teams must rely upon networked access (Arpanet, USC-LAN, uucp/bitnet, and kermit-based telephone) to SF processors for electronic mail, file transfer, remote login, software development, and interactive consultations.

on the SF approach to LSSE.

2. Outstanding Problems in Large-Scale Software Engineering

Developing large software systems in a highly productive, high quality, and cost-effective manner will be a major challenge in the 1990's for all large high technology, engineering, and manufacturing firms. We believe that new or improved tools for computer-aided software engineering by themselves are not the most effective strategy for meeting this challenge. Instead, we hold that improved software productivity, quality, and long-term cost reduction must integrate advanced software technologies with techniques for their use by large changing staffs who will be trained and managed within specific organizational settings [51, 42, 20, 45, 12, 50, 49, 3].

Large-scale software systems (LSS) typically represent a substantial collection of programs that total between 25K-500K lines of source code. These systems are developed by teams of 10-200 or more programmers, intended to be used for many years, and maintained by generations of new staff.[3] LSS therefore represent dynamic open systems that are never completely understood in detail by any individual, as well as other problematic challenges [19].

LSS development projects are people and resource intensive. The time, skill, and commitment of software developers and managers are critical resources that a development organization must effectively mobilize and sustain. Organizing system development activities with available staff and resources is a precarious endeavor. Incremental reorganization of these activities is ongoing in response to staff turnover, changing system requirements, acquisition of new system upgrades, shifting occupational or career advancement contingencies, ineffective system development methods, shifting budget and schedule constraints, and other changing circumstances. When project organization cannot effectively be matched to the tasks needed for system development, then costly problems such as work slippages, errorful systems, or mis-timed system logic can appear. Subsequently, the problems of producing large computing systems will primarily be organizational. These problems will center around how to organize available staff and development resources in an organizational setting to engineer software products throughout their life cycle, and in ways that seek to achieve high rates of productivity while mitigating against the introduction of errors. As such, solutions must also be of an organizational character if they are to be most effective.

Our objective is to bring together a set of automated software engineering tools, flexible software development techniques, and project management strategies together with a large staff and computing resources into an experimental organizational form we call the System Factory. Within the System Factory, we develop LSS in ways that allow us to encounter the various organizational problems that emerge, and endeavor to find effective solutions. In particular, we have focussed our attention to the following problems of LSSE that include how to:

- *Construct computer-aided software engineering environments* that can be configured to support all or part of the system life cycle;
- *Specify reusable software* concepts, artifacts, and programs;
- *Manage and train* staff subject to frequent turnover;
- *Support the evolution* of multi-version software product families;
- *Support exploratory development* for rapidly prototyping emerging software concepts or technological innovations;
- *Integrate* an open system of tools, techniques, management strategies, staff skills, and other resources to support LSSE work in different organizational settings.

In our experience, each of these problems of LSSE are continually encountered by individual software

[3] In the U.S., most LSS staff stay on a job typically 12-30 months. LSS that exist for three years or more usually must survive one or more cycles of complete staff turnover.

engineers, software engineering work groups, and interacting networks of software teams working on LSS projects. Since each must confront the problems and find solutions, then such solutions must be coordinated across all work units to be most effective. The System Factory project thus serves as an appropriate experimental laboratory for studying these problems and affecting solutions or interventions.

The following section describes the history of the SF project.

3. Genesis and History of the System Factory Project

The SF is an investigation into LSSE with large staff in an academic setting begun in 1981. At that time, few studies of software engineering project courses or LSSE projects had been published. But a growing number of studies of the evolution of new computing technologies and large computing systems in complex organizational settings were available [23, 24, 25, 38, 26]. These studies indicated that the development and use of large systems was plagued by a background of recurring dilemmas that diminished the potential benefits, decreased productivity, and raised the cost of system development and use. These studies found that the structure and function of computing systems was inextricably bound (or "webbed") to the organizational settings where they were produced and consumed, and to the jobs, careers, and circumstantial interests of the people who animated them. This meant that if we were to engage in a LSSE project, we needed to investigate not only the role of new software tools and techniques, but also how the project's setting would interact with the system products being developed, how they would be developed, and who would be doing the development work. As such, if the interaction was benign, then the new software technology might be most effective. On the other hand, if the interaction was substantial, then we could expect to encounter problematic situations that could decrease development staff productivity, reduce product quality, or otherwise raise the cost of LSSE.

The ideal candidate for such a study would therefore be a multi-year, LSSE project that would require a complex organization of people and computing resources situated within some larger institutional context. A long-term project would assure a dynamic project organization in terms of staff turnover and innovations in local computing facilities. A LSSE project would inherently require a large staff, extensive use of available computing resources, schedules, production plans, administrative controls, and software engineering tools and techniques. The larger institutional context would create a marketplace of occupational and career contingencies for project staff, of external administrative units to manage base computing facilities and provide support staff, of extramural research and development funds, and of computing system vendors to provide upgrades to local computing facilities. Finally, the LSS system to be developed should be unfamiliar to allow us to experience the uncertainty in the final shape of things to come, so that we could try, fail, learn, and manage as we go. The SF would therefore be an experiment that represents a complex, real-world LSSE project based in an academic setting.

3.1. Initial Conditions of the SF

Our objective was to mobilize available staff and computing resources to develop a language-independent software engineering environment (LISEE) [39] within schedule, budget, and computing resource constraints. The LISEE would initially be the LSS software we would develop in the SF.[4] If successful, we could in principle then employ the LISEE tool ensemble in other LSSE projects.

In January 1981, we started with a development staff of 57 graduate software engineering students who elected to take a 15-week semester course in LSSE. These students were competent small-scale Pascal programmers possessing an undergraduate degree in computer science or the equivalent, but generally assumed to lack prior skills in LSSE. Students were expected to commit 10-12 hours per week to this course. Since this was a course televised from USC to local industrial settings, this meant there was a partial geographic distribution of students in 5 different remote TV centers in addition to the majority of in-house students. There was a visible ethnic diversity of students represented by at least 10 different cultural or national backgrounds. At least one-third of the students spoke English as a second language.

---

[4]This LISEE eventually evolved into the SF's current software engineering environment described in a latter section.

All software development was to be performed on a centralized, time-shared DECsystem-10 mainframe system with minimal programming support environment. There was a firm schedule for software delivery (end of the semester deadline is absolute) and budget (no discretionary funds available) in place. We then started with a modestly articulated model of software engineering technology (as of 1981) as found through a comprehensive literature review and reading list 12 pages long, with about 200 citations, prepared by the author.

Following a number of introductory lectures, we provided the staff with the software technology reading list partitioned into general reference materials and potential LISEE components. This initiated the beginning of the project. Next, we randomly assigned small number of students to review selected reference materials and become knowledgeable about one domain of software tools pertinent to the LISEE. Example domains included structure-oriented editors, testing systems, database management systems, user interfaces, etc. [39]. We then provided the staff with initial development requirements in the form of a conceptual LISEE architecture. This architecture served as the basis for dividing project staff into project teams of 2-7 cooperating students. Ten teams were established, five in-house and one at each remote site. Each team was then responsible for developing one component tool of the LISEE. We followed by providing the staff with techniques for developing the LISEE. These techniques addressed the the conventional stages of the system life cycle: requirements analysis, functional specification, architectural design, detailed design, implementation and integration, testing, user documentation, and maintenance. Basic background in these techniques was derived from examples available in the literature at that time (cf. [42]). Last, we provided the staff with background lectures on the organizational problems and strategies of LSSE as derived and transformed from the previously cited studies of the consumption of computing systems in complex settings.

Actual LISEE development was scheduled for eight weeks, following seven weeks of introductory lectures and background preparation (readings, class discussions, homework assignments, exams, and informal out-of-class discussions). The project's development was scheduled to reiterate the techniques, problems, and strategies for performing one system development life cycle stage each week, while staff performed that life cycle activity [42]. Thus we could use class time to discuss problems that different teams encountered as examples during that system development life cycle stage. Of course, we did not know if this was reasonable or if it would work overall. But we were there to learn through success or failure. After the project's eight week development, all ten components of the LISEE were prototyped, demonstrated, documented, and delivered by the staff of then 53 graduate students.

3.2. Outcomes and Implications of the Initial SF Experience

Version 1 of the LISEE represented 30K+ lines of Pascal code operating on a TOPS-10 based DECsystem-10 mainframe. The tools developed ranged in size from roughly 1500 to 4000 lines of code. All LISEE components were demonstrated to be operational, although there was variation in the quality and amount of system development work completed by the different SF teams. Each LISEE tool provided an operational interface or stub for interconnecting to at least one other component in the LISEE architecture. This LISEE was clearly not production-quality, but our objective in the SF experiment was to demonstrate that a LSS system of the complexity of a LISEE could be prototyped by a large staff in an academic setting in a relatively short time.

Each team submitted project team documentation that recorded their work completed in each system life cycle stage. There were eight chapters to each team's documentation, one chapter per life cycle stage. On average, each team delivered 50-100 pages of system life cycle documentation per person. That is not to suggest that every person produced that volume of documentation, but rather that a two-person team might produce 150 page project document, while a five person team might produce a 300 page project document. In total, about 3000 pages of LISEE development life cycle documentation were delivered.

Overall, the general feeling among all project participants was that the project was a success, and that most students valued the software tools and project documentation they developed. For a number of students, this was the most substantial and best engineered piece of software they had yet developed,

and the largest group project in which they had ever participated.

3.3. SF Iterations: 1982-1987+

Although the initial SF project was successful, maybe we were just lucky or misleading ourselves. That is, could the experiment be replicated with an entirely different staff and produce comparable results? In order to answer this question, we decided to repeat the experiment with the same objectives but with entirely new staff and same computing environment. However, this time a smaller staff of 30 undergraduate students was employed using same computing facilities as before, and same development techniques and project management stretegies followed. Roughly comparable results were produced in terms of a smaller, less ambitious LISEE and related documentation.[5] This repetition of the SF experiment indicated to us that the SF concept was sufficiently viable for further experimentation, refinement, and development. Since then, we have repeated the SF experiment in an iterative manner, incorporating new refinements, insights, and technological enhancements. Some highlights follow.

In the next iteration, we chose to utilize the first generation SF documentation and software as prototype available for reuse if desired by team members. We revised the basic LISEE architecture, the functional capabilities of its tools, the software life cycle development techniques, and the project management strategies to better accomodate defiencies observed in the initial iteration. Each subsequent iteration would also give rise to a revised set of tools and architecture, techniques, and strategies.[6] Also, the computing environment was upgraded to TOPS-20 operating system on the same computer system. This was not our decision, but it happened in our work setting, thus we had to transition to it in order to continue.

Next iteration, we were given another opportunity to migrate to a new computing environment, this time to a VAX-VMS system. We also decided to migrate the LISEE from Pascal to C, and to expand the scope of the LISEE to support experiments in VLSI circuit design [40, 20, 41]. However, as our experience, teaching materials, and software tools expanded, we came to find the seven week introduction, and eight week development schedule too confining. The choice we opted to implement was to expand the one semester course into a two semester, academic year length course. This would allow us to take more time and explore at greater levels of detail, the tools, techniques, and strategies we were putting into practice. This also would be a good time to again migrate, revise, and redevelop the LISEE (now called simply a SEE) in C to a VAX-750 running Unix 4.2bsd [44]. This migration then represented the next iteration. Finally, for the current iteration, the SF was expanded to utilize a loosely coupled network of heterogeneous computers. The SEE was continued in C and C++ on two VAX-Unix 4.3bsd systems and two SUN-3 workstations all on a common LAN, as well as migrated to operate on a separate network of AT&T 3B2 and 3B20 computers[7] all running Unix System V. A TI Explorer Lisp Machine was made available, as were a number of IBM-PCs running MS-DOS. These latter machines were used for prototyping next generation knowledge-based SF tools, document preparation, development of small program modules, and remote file transfer.

In sum, we refined and redeveloped the SF's tools, techniques, and strategies through seven iterations. In the course, we provided hands-on training with the development, use, and evolution of the SF's technologies for more than 500 graduate students in total over the seven year period. We now turn to describe the SF approach as it now stands in terms of the SEE, software life cycle development techniques, and project management strategies we employ.

---

[5]Of course this is not a true replication of the original experiment. But we felt this case study experiment was a close approximation of the first, and therefore comparable.

[6]For example, the original LISEE required a relational data base management system to be used as a central archive of project related information. Development of a RDBMS was then started and continued until the SF migrated to a Unix 4.2bsd environment, where we were able to use the existing Ingres RDBMS, a system more capable than ours. Continuing development our the original SF RDBMS was then halted, and another new software tool was added to the SEE for ongoing development.

[7]These computers have since been decommissioned and replaced with a network of SUN-3 workstations.

The next section describes selected results we have produced in tackling these problems in the SF approach to LSSE.

4. The System Factory Approach: Product, Process, and Setting

Central to the SF approach to software engineering is a joint focus on three key determinants of the outcomes of LSSE: the *products* developed, the *process* through which the products are developed, and the *production setting* where the process creating the products occurs. The SF's products embody a technological structure and function.[8] The software production process, whether organized according to a "waterfall model" or prototyping-incremental development system life cycle must be planned, staffed, directed, scheduled, budgeted, and controlled to accomodate smooth production of the intended technological products by available staff. Formal software development techniques may be used to further structure this process. The production process therefore serves to structure the flow and transformation of organizational resources into technological products and work arrangements.[9] Finally, the production setting provides the staff and resources that are mobilized according to organizational policies, procedures, histories, incentives, and pressures in its marketplace to animate the process of product manufacture. In particular, we find the organizational and technological arrangements that support software production often have a profound affect in shaping how software development occurs. For example, the base computing environment used in the SF was changed seven times (once each annual project cycle) primarily due to actions of the university's computing facility through their attempt to provide modern computing services that are easier to sustain and operate. However, we find that many researchers seem to ignore the importance of how idiosyncratic features of each setting uniquely influence how computing resources are consumed, how software systems are produced, and what products are produced.

Clearly there has been increasing attention directed in the software community as to whether LSSE depends more upon the nature of the product or the process. However, little attention is directed at the organizational setting where a particular group of people must work together using available resources to develop software system products according to some formal or ad hoc development process. Academic computer science departments, computer system manufacturers, aerospace contractors, banks and insurance companies, and national scientific laboratories represent some of the places where LSSE occurs. Clearly there are differences across and within such settings in terms of the kinds of software applications developed, development tools employed, computers and operating systems utilized, programming languages primarily used, professional background and skill level of the software developers, budget and development schedule constraints, and so forth. As such, we find the influence of the organizational setting on the products produced, the process through which they are produced, and the joint influence of each on the other is fundamental [26, 42]. In our view, overlooking these patterns of influence is thus a fundamental mistake in attempting to understand how LSSE occurs.

This section therefore describes selected products, production process techniques, and project management strategies used in the SF. After this, we describe a potential future application of SF technology. We then describe our experiences in transferring these SF technologies to other organizational settings.

---

[8]The SF's products include programs, documentation, software development techniques and analyses, application-domain knowledge, routines for SF use and evolution, strategies for managing LSSE, and people trained and skilled in LSSE. In addition, a number of research publications have been produced, presented, and disseminated.

[9]Elsewhere we refer this transformation of organizational resources into technology-based work arrangements as "packaging", and the ensemble of products a "computing package" [25, 38, 26].

4.1. SF Product: A Computer-Aided Software Engineering Environment

The SF's CASEE represents a reconfigurable emsemble of software tool components. It is designed to support the computer-aided engineering of VLSI or software systems throughout their life cycle [20, 40, 41, 44]. However, our focus in this report is limited to large software systems. The SF's CASEE represents a LSS consisting of more than 250K source code statements and 5K pages of processable documentation. It is designed to process the family of languages that describe each life cycle activity. For instance, one way this is realized is by generating and configuring a set of language-directed tools that can evaluate language-based system descriptions for consistency and completeness. Another way is to generate and configure a set of tool components to serve as the base for specifying, rapidly prototyping, and validating particular application systems. Accordingly, we built the CASEE on top of, and integrated with, available operating system environments (Unix 4.3bsd, Unix System V, MS-DOS, and Lisp) that operate on our network of processors. The CASEE tools constitute the System Factory's production infrastructure of software machinery used for fabricating and analyzing software system descriptions, as portrayed in Figure 4-1.

What follows is a brief description of the System Factory CASEE tools that we are investigating:

Language-directed editor generator: utilizes formal language specifications to produce a language-specific editing environment that detects syntax errors and inconsistencies in type declarations and construct usage at keystroke entry time. Lanugage-directed editors for software specification, design, coding, and animation languages have been generated and put into to use in the SF. It also constructs an abstract syntax tree and symbol table that can be used as inputs to the testing system and Gist specification processor. We have used this tool to rapidly construct a facility for generating formal system specifications from table-structured input of narrative system requirements. This was done by restructuring LDEG to accept object-oriented (semantic network) specifications, thus becoming a kind of knowledge-directed editing environment [57].

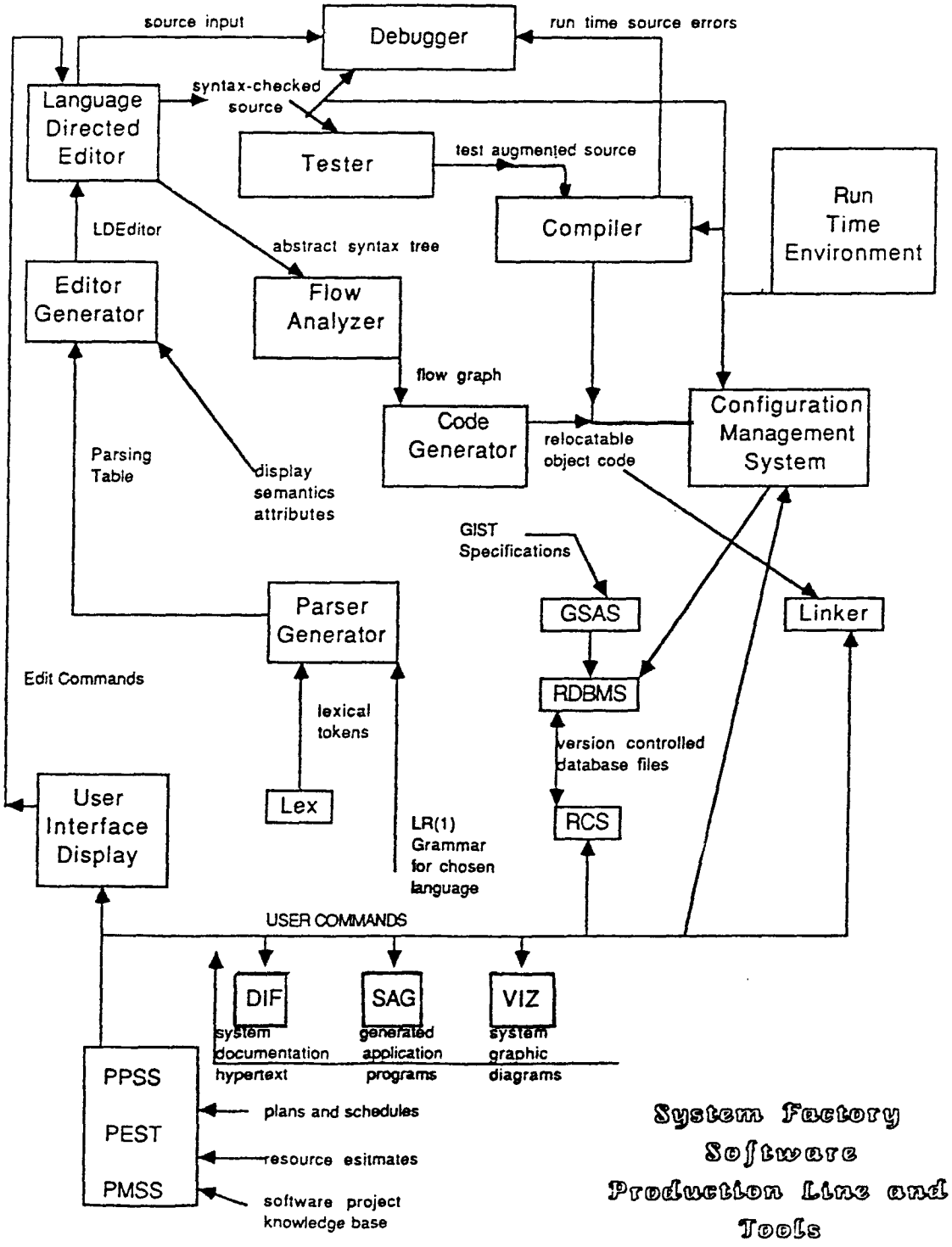
Flow analyzer and testing system: performs static control flow and dynamic data flow analysis of source code descriptions, produces augmented source programs with probes for execution monitoring and debugging, and provides a rule-based mechanism for generating test cases for software modules interfaced to a debugger. This tool supports validation of a software application's performance requirements. We intend to integrate this testing system with the CMS to support a knowledge-based tester of multi-version system configurations.

Configuration management system: (CMS) provides automated mechanisms for controlling multi-version system descriptions (e.g., specifications and source code implementations), through use of a module interface and interconnection definition language (NuMIL) and compiler [31, 30]. The NuMIL language and processing environment are used to specify the architectural configuration of new applications, or used to generate such a specification for existing applications. These specifications coupled to their source code implementations provide an appropriate medium for maintaining large evolving software applications [31, 30]. This system is currently being extended to support the design and configuration management of distributed, real-time multi-processor systems. A new facility has been designed to help coordinate and communicate to staff the modification of configured system components for large development projects [21, 53]. We intend to further extend this system to incorporate mechanisms for reasoning about alterations made to the structure and function of software system configurations.

Window-based user interface and command processor: provides a window-based command/shell processor, mail handler, and display manager that other tools or application systems can use as their "front end." We are currently incorporating an additional mechanism for creating active display devices (real-time gauges and digimeters).

Gist specification analyzer and simulator: provides a basic facility for constructing, analyzing, and functionally simulating system specifications written in the Gist specification language [1, 8, 46]. We intend to extend this facility to support mechanisms for explaining the behavior of operational system specifications. These mechanisms are being developed in conjunction with the FSD environment at USC/ISI [2].

Figure 4-1: The System Factory SEE



Document integration facility: (DIF) provides a facility for creating and maintaining a multi-version electronic encyclopedia of system documentation hypertext [58, 27, 15]. This enables us to interrelate and trace textual/graphic system documentation across all system life cycle activities that can be uniformly queried, browsed, revised, and archived. Life cycle product descriptions for all SF CASEE tools are currently included in the DIF database archive. We intend to extend DIF by integrating it with the CMS to document multi-version system configurations, with SAG and PPSS to support electronic project management documents, and VIZ to create animated documents that visualize system features in-the-large and in-the-small.

Spreadsheet application generator: (SAG) provides a special-purpose environment for specifying and rapidly generating interactive spreadsheet-like application programs. SAG has been used, for example, to develop a modest decision support system (PEST) for developing software cost models and calculating computer system acquistion costs. SAG is interfaced to a relational data base management system so that it can utilize data bases created by other tools (e.g., PPSS). We are currently experimenting with techniques for generating rule-based spreadsheet applications [10], as well as non-rectangular, n-dimensional, distributed, and dynamically reconfigurable spreadsheet applications. Potential applications of this kind include simulation of dynamic systems, cellular automata, distributed intelligent system shells, and emulation of reconfigurable supercomputer system architectures [18, 52].

Project planning and scheduling system: (PPSS) provides a rule-driven data base system for planning project schedules, work breakdown structures, task precedence networks and diagrams, and their interdependence (cf. [14]). These mechanisms can be employed to establish and assess the impact of changing requirements on a development project's budget, schedule, and production plan. We are currently integrating the services of this system into those provided by the PMSS.

Knowledge-based project management support system: (PMSS) a knowledge-based software process support system (cf. [37]) that incorporates an operational model of the software life cycle processes embodied in KBSF development techniques and project management strategies [12, 50]. PMSS is a kind of intelligent system that represents, simulates, and reasons about the software development process. The initial demonstration version of PMSS was implemented by manually transforming Gist specifications of selected software development subprocesses into an OPS5 rule-based system. However, the current version of PMSS is being constructed using the Knowledge Craft[10] expert system development environment.

System prototyping and dynamic visualization system: (VIZ) a visual programming, system prototyping, and diagram building facility [13] that supports the visual specification and script-based animation of 2D/3D software system descriptions [36, 28]. We are currently extending this system by integrating it with the CMS to produce multi-dimensional (2D/3D color graphics) layouts of static and dynamic configurations of multi-version systems, and system components. (These layouts resemble the complex circuit diagrams used to design, animate, debug, and document VLSI circuits.)

Unix (lex, yacc, ingres, rcs, mm, troff, curses, more, OPS5, emacs,...): many of the language development, relational data base management, revision control, and related tools available in the Unix operating system serve as a foundation for most of the CASEE tools listed above.

4.2. SF Production Techniques for Software Life Cycle Engineering

Many tools in the System Factory CASEE are language-directed. This means we can configure and generate a software tool set that can process alternative language-based descriptions for a particular system engineering application.[11] It is possible to specialize a family of tools supporting system descriptions occurring at each software life cycle stage. Using this approach, system life cycle product

---

[10]Trademark of Carnegie Group Inc.

[11]For example, this suggests that it is possible to generate a tool set that can check/enforce local software quality assurance standards (e.g., follows certain design rules) when encoded in the (attributed) language specification used for tool initialization.

descriptions such as software requirements, functional specifications, architectural configuration specifications, detailed module specifications, user-system dialogues, executable source codes, test case specifications, maintenance procedures, and structured diagrams can be processed as long as they can be described via the LR(1) language specification formalisms we use. Subsequently, these kinds of language-based descriptions are also amenable to automated manipulation supporting system evolution [31, 30].

Thus a central facet of the SF techniques for engineering software systems throughout their life cycle is to specify each stage of development in a formal, processable language. Processing tools can then be configured in ways that check the consistency and completeness of a given specification, and thereby reinforce the the specification technique appropriate to each stage of system development. In addition, such SF techniques must incorporate strategies for incrementally developing software descriptions in ways that utilize available processing tools.

For instance, consider the intermediate stage of system development where a software system's architectural configuration must be specified. A technique for specifying the architectural configuration (or structure) of a LSS requires identifying a network of operational modules that progressively transform imported objects into exportable data resources. The portals through which imports and exports move are the module interfaces. But structural specification first requires partioning the system's functional specifications into realizable functional components. Accordingly, we must choose between alternative partitions of software components, depending on whether such components are readily available (reusable) or must be built. The selected partition then circumscribes decisions for allocating computing resources and staff to module development. As such, the structural system specification defines the boundaries for distributing concurrent computation across the system, as well as dividing the detailed design and coding work among available development staff. Thus, if the division of work changes (due to staff turnover for example), then the system's configuration may evolve, and vice versa.

Since LSS configurations inevitably evolve over time, then there is need to describe the evolving system architecture in a form whose consistency and completeness can be checked and maintained. Further, in LSSE projects, dispersed groups of developers may specify, code, test, and modify different modules asynchronously. This means that a given module might exist in a number of different but related versions at any time. It also indicates that insuring configuration integrity is a major problem in LSSE project coordination. Similarly, the upward-compatibility of a modified module or subsystem with respect to the rest of the system (and related component families) must be checked for consistency [32]. Subsequently, LSSE projects can benefit from a *system architecture specification processor, module interconnection and interface definition language*, and *configuration diagram visualizer* for managing families of multi-version modules. The module interconnection language is the medium for specifing system architectural design and configuration [33]. However, we require a MIL that can represent families of multi-version modules and subsystems. In our case, we developed and employ NuMIL, a MIL designed with these requirements in mind [30]. An example in the following figure shows a NuMIL specification of a family of subsystems with different processor versions, each composed of different versions of two common modules.

Overall, these processing requirements represent what the SF CASEE *configuration management system* provides [31, 32, 30], as portrayed in the last figure. Clearly, such a CMS benefits by incorporating UNIX utilities such as `make, sccs/rcs`. A *relational database management system* (e.g., Ingres, Unify, or Oracle) allows system developers to store, browse, and query information about families of configured system components, and to control concurrent access to these components. However, such RDBMs require a separate object-oriented interface to properly map the descriptions of configured system components into a relational format [30].

4.3. Strategies for Managing Large Software Projects in the SF

How do we manage a large engineering team to develop, use, and evolve a LSS system? That of course is the question we face. But we believe we have a unique approach to investigating this important question.

```
subsystem S is
   provide a,b;
   require c,d;
   configurations
   /*** Subsystem S has two configurations IBM-PC and VAX ***/
      IBM-PC = { M_1 : version_1, M_2 : version_2 ;}
      VAX = { M_2 : version_1, M_1 : version_2 :}
end
```

```
module M_1 is
   provide a, foo;
   require d, b;
   implementations {
   /* M_1 has two versions */
      version version_1 {
         realization x.c;
            provide
            int a;
            short foo;
            require b(), d;}
      version version_2 {
         realization y.c;
            provide
            float a;
            int foo;
            require b(), d;}}
end
```

```
module M_2 is
   provide b;
   require c, foo;
   implementations {
   /* M_2 has two versions */
      version version_1 {
         realization m.c;
            provide
            int b(s,t) char *s, *t;
            require c, foo;}
      version version_2 {
         realization n.c;
            provide
            int b(m,n) char *m, *n;
            require c, foo; }}
end
```

Figure 4-2: Example NuMIL Specification

Since the late 1970's, we have been conducting in-depth field studies of large system development projects in both industrial and academic settings.[12] Through careful comparative analysis of many engineering projects, we are systematically identifying problems of organizing and managing these projects as well as the strategies used to handle these problems. Rather than describe the analysis and the problems, we instead outline five strategies for managing large software engineering projects.[13] We also intend that some of these strategies might evoke an awareness of similiarity to readers who have observed or practiced such strategies without seeing them made explicit.

4.3.1. Focus on the coordination of development work and workers

One set of concepts found useful for effective project organization emphasize the importance of establishing a *socially proactive, democratic workplace* [4] intended to encourage *computer-supported cooperative work* [6, 16, 17, 50, 3]. Accordingly, the particular concepts to follow include:

- provide avenues for both system developers and eventual users to participate in the decisions determining the system's features, purpose, and modes of use;
- openly share strategic information regarding the purpose, intended uses, and expected outcomes with participants developing and using the system;
- provide for a fair sharing of the benefits between participants arising from both the development and use of a new system;

---

[12]This includes a sustained effort at studying the dynamics of the System Factory project itself.

[13]The reader interested in the studies, analyses, and problems should consult [26, 51, 42, 43] [50, 16, 17, 3]. Also consider [22] for an introduction to the problems in a similar setting.

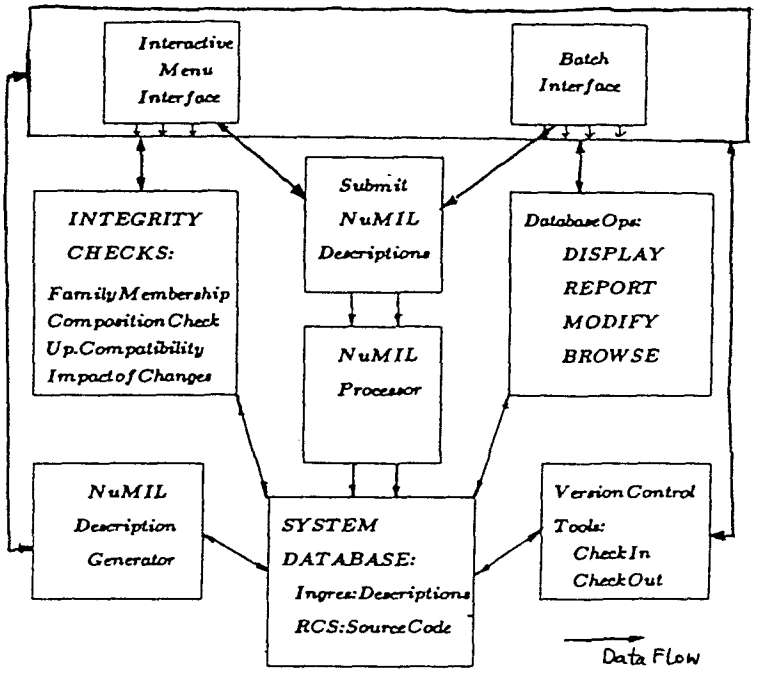


Figure 4-3: The SF Config. Mgmt. System

- guarantee participants will not be penalized for speaking out or criticizing proposed directions for local system development efforts;
- since there may be disputes between (or among) system developers, managers, and users, provide an organizational mechanism for a fair settling of those disputes;
- encourage a participatory, democratic awareness among system developers, managers, and users so that they will be committed to accomplish the best job possible with available resources, and to work through (or around) the problems that arise in developing systems.
- continually seek to provide new tools and techniques that facilitate the conceptualization and collaboration of system development activities by a large staff working in a complex organizational setting.

These concepts thus provide a basis for the other strategies for organizing system development work that follow.

4.3.2. Design project organization to facilitate commitment

Everyone will be responsible for managing some portion of overall work activities. Since project managers will be responsible for coordinating work and resources within the local computing infrastructure, they need to know about bottlenecks and other troublesome conditions. Maintaining a high-level of software production requires the commitment of staff and resources to achieve it. Continuity of staff commitment to project and colleagial objectives is more important than mangerial control over

these staff [38, 26]. Management control in large projects is distributed and more subject to contention. Project staff who must be coerced into performing undesired tasks cannot be expected to perform those tasks with high productivity and care. Maintaining staff commitment requires regularly assessing the conditions that bind their commitment to work including desired resource availability, local (dis)incentives, and career opportunities. Such an assessment emerges when staff participate in deciding how to realize project objectives. The regularity of assessment depends on the perceived stability or uncertainty of local project management conditions. Unexpected circumstances will always emerge and give rise to destabilizing conditions. However, strong commitment will often provide staff members idiosyncratic motivation to accommodate local contingencies until the prevailing order is re-established, unless their commitment is sufficiently weakened.[14] But if their commitment to project objectives is strong, so that their perceived investment (or stake) in project activities is clear, then they can build on their investment by discovering new ways to perform their work [3].

4.3.3. Identify the incentives that motivate project participants

Why are software developers as productive as they are? This may seem to be an odd question, since many people are fond of asking how do we make software developers more productive. But we find that to answer the latter, you must first be able to answer the former. In particular, we find that what motivates software developers to be as productive as they can be is often idiosyncratic and circumstantial [38, 26]. What motivates developers today, may not motivate them equally well tomorrow. For example, in working with a student staff in the SF, one might assume that they seek to be very productive to insure earning a high grade. This seems true for some staff, but not all. Some students seek to gain first-hand experience and research skill in the development and use of state-of-the-art software tools. Other students hope for advancement or promotion in their job, based on their acquisition of certain new technical skills, such as LSSE. Still other students have an entreprenuerial spirit, and hope to identify new software or services through their participation that they could eventually develop and profit from in the private sector. Finally, many students are motivated by more than one of these situations. In sum, near-term rewards such as a high grade may provide an incentive for certain staff, while other staff are primarily concerned with long-term professional, occupational, or financial opportunities. Therefore, in the SF, we seek to regularly assess and cultivate the incentives that might provide intrinsic motives for each staff participant to be highly productive and quality conscious.

4.3.4. Develop new software technology as a package

LSS systems are more than a simple collection of many programs and source code files. Every large software system assumes some configuration of hardware, telecommunications facilities, software base, documentation, time, money, skills, organizational units, management attention, and other resources for its productive development or use [24]. This "package" of computing resources outlines a set of requirements that must be met by participants working within the local computing infrastructure. Each package must fit transferred, inserted, and transitioned into an idiosyncratic local setting. As such, users need to know what resource requirements are built into a new technology in order to assess both the costs and ease with which it can be transferred into existing computing arrangements. However, as a new technology is transferred and assimilated into ongoing organizational routines, the local computing infrastructure will be altered to reflect its repackaging. Historical trends in software engineering indicate that this repackaging is done to make the work activities more productive and routine. However, these trends also indicate a finer division and specialization of labor among participants as well as increase the number of resources to be coordinated. Accordingly, an important cost of using new software technologies intended to make the work activities more productive is increased demands for attention to detail, coordination, and routine. This is a form of work management that individual participants must increasingly perform. Thus, in developing a new software systems, tools, or engineering methodology, a strategy for managing its life cycle in its target setting must explicitly be built into its package to facilitate its transfer, insertion and transition [38].

---

[14]The building of strong commitment for some ("signing up"), and the weakening of commitment for others ("burning out") is a key element in what gives a new system its soul [22].

4.3.5. Organize the SF to Support Flexible Software Manufacturing

New forms of manufacturing supported by flexible manufacturing systems (FMS) and computer integrated manufacturing systems utilize interchangeable (reusable) components, standardization, automated production processes in order to build products in an increasingly efficient manner [35, 34]. FMS also require production staffs who must be reasonably well organized, managed, and coordinated as well as committed to producing high-quality products. Further, these forms of manufacturing are more amenable to higher levels of knowledge-based automation [14], and represent alternative forms for organizing software production.

A FMS consists of production workstations connected by an automated workpart-handling system [34, 35]. A *flexible software manufacturing system* (FSMS) would also. A FSMS can process various different part types using production process planning and group (part/component family) technology principles [7]. A FSMS supports (1) the specification, design, and fabrication of component families, (2) asynchronous introduction of components into the production process, (3) reduced manufacturing lead-time, (4) reduced work-in-progress inventory, (5) increased workstation utilization, and (6) redirected skill enrichment.[15] Skill requirements with a FSMS can then shift toward loading/unloading raw/finished components, changing tool and workstation configurations, diagnosing and repairing errant component processing or tool functionality, reprogramming (i.e., specifying new) production process sequences, and sustaining and evolving the computing resource environment.

We are currently exploring alternative ways of organizing our software production process, where the SF CASEE serves as the flexible base of reconfigurable production machinery and component-handling system. The production resources currently in the System Factory consist of a emerging collection of knowledge-based software system specifications, a staff of 30-70 graduate student developers available for up to nine months at a time, the CASEE components, and a network of DEC VAX-750 (Unix 4.3bsd), AT&T 3B20 (Unix System V), IBM PC (MS-DOS), SUN (Unix 4.3bsd) and TI Explorer (Lisp) processors. The knowledge bases describe in formal or narrative languages, the operational and non-operational requirements of 20 of the most common software system families[16] and complete system life cycle descriptions for the CASEE tools. The FSMS production process planning, scheduling, and management system (PMSS) is, however, still very experimental and currently being redeveloped [50].

This FSMS approach still requires more refinement. As such, we must continue to identify what kinds of work, organizational, and resource arrangements are fundamental to the development and use of a flexible software manufacturing system.

4.4. Understanding and Articulating Software Management Strategies

What these five strategies describe are inherently organizational approaches to more effectively deal with the problems of large-scale software engineering project management. These strategies are quite different than those derived from a traditional software engineering project management approach [11, 55, 5, 21]. Elsewhere, we identify a dozen additional strategies for handling other problems in organizing and managing system engineering projects with large staffs [26, 51, 42, 43, 3]. However, as we continue to investigate and apply these strategies, we expect that they will be refined, yet kept practical. This is one of the goals of our research. But to reach it, we must be able to codify our emerging knowledge base of project management strategies, preconditions, and consequences to allow this research to continue over a long-term period with changing engineering teams.

Accordingly, we recently began an effort to codify our knowledge of the software production process,

---

[15]As Burbridge observes, the introduction of group technology in manufacturing accomodates an alternative division of labor that can enrich the skills of manufacturing specialists, and thereby enable their increased effectiveness and satisfaction with their work [7].

[16]This information was articulated and analyzed from over 400 reference materials by students participating in the System Factory project [47]. We intend to use this knowledge of the software families to articulate reusable specifications of these system families in Gist [1, 46].

the System Factory development techniques, and the strategies for managing software development projects into a knowledge base system (PMSS), using GIST as our knowledge specification language [50], and Knowledge Craft as our implementation environment. Our intent is that when this knowledge base is viewed as an operational specification of the software production process, we will have an important new medium for communicating, simulating [1, 9], visualizing (cf. VIZ), and explaining [54] knowledge of how to organize the development of LSS in ways sensitive to local organizational dynamics and system engineering practices. Putting this knowledge into a computational form will allow us to evolve and evaluate the knowledge base as we acquire new information about software projects conducted in different organizational settings. This will enable us to create a comparative "corporate memory" of the process of LSS development. Also, as we have observed elsewhere, such a knowledge base provides a foundation for simulations of cause-effect relationships due to the adoption of potential software productivity aids, and other forms of software technology transfer [49, 48]. We intend to put this knowledge base and delivery mechanism into practice within the System Factory.

5. Potential Future Applications

One way to convey for potential of the SF approach to LSSE is to describe how it could be applied to next-generation system development project. Since what might constitute a next-generation system is vague and speculative, consider the following hypothetical scenario: Suppose our task is to R&D a new kind of computing system, involving multiple processing elements, with each processing element being constructed from wafer-scale or some other ULSI circuit technology. In addition, such a multiprocessor engine, while compatible with existing operating system mechanisms, must support selected software applications and software development tools. These requirements therefore imply that a large-scale development effort is required, involving 50-60 hardware and software engineers.

In developing the hardware circuit components, we face a number of tasks. Wafer-scale circuits must be manufactured using materials and fabrication recipes different from those currently in use. This implies we must conduct materials or fabrication process simulations, to make sure the circuits produced will possess the expected device physics and electrical properties. We thus choose use SAG to rapidly contruct a process simulation program to help articulate electrical circuit properties, as has been suggested elsewhere [18]. Next, since we will be designing high-complexity circuits, we require an ensemble of CAD tools.

In developing complex circuits, we must be able to specifying alternative versions of circuit behavior, stucture, and cell logic design. From our viewpoint, we can use Gist to specify circuits input-output behavioral requirements, at varying levels of details and specificity. All the while, we can evaluate circuit specifications for consistency and completeness, as well as simulate their developing functionality, albeit slowly. This allows for early testing and validation of circuit design behavior before fabrication.

As circuit functionality begins to densify, we then start to partition and configure circuit functionality so that control, data flow, and timing signals can circulate appropriately. To insure this, the partitioned functional cells and their interfaces are carefully placed, then interconnected in NuMIL specifications. Since these specifications are still symbolic, they maintain sufficient elasticity to allow (or encourage) creative cell logic design. However, since many circuit designers are skilled in graphic logic design, they should be able to interactively visualize (Cf. VIZ) emerging circuit or cell design with graphic elements or icons. Nonetheless, as long as the graphic and textual descriptions of circuit specification share a common semantic interpretation (via the same intermediate representation), then either descriptions will be suitable for evaluation by CASEE specification processors.

In concert with recent developments in CAD research, we decide to employ the VHSIC hardware description language (VHDL) [29] for "silicon programming." Subsequently, we then generate a VHDL-directed editing, testing, and compilation environment using CASEE tools (cf. [20, 41]). These tools are specialized in such a way that the editor detects circuit design rule violations as flow graph errors in a VHDL description. Design rule checking can therefore be performed essentially at VHDL code keystroking time, well before compilation and layout. Similarly, the VHDL compiler's code generator must be specialized to emit low-level codes corresponding to circuit mask geometry layout. The VHDL testing

system also outputs a version of a circuit's program that can be tested before or after fabrication, for validating individual and assembled cells logic, structure, and behavior. In addition, a circuit configuration management system would be specialized to store and maintaine the integrity of these evolving, multi-version circuit descriptions. Finally, since evolutionary changes in circuit behavior, structure, or logic design will occur, then their language-based descriptions can be processed by the appropriate CASEE tools to determine where potential inconsistencies may occur. In turn, this should help reduce the costs of circuit design maintenance.

In parallel to circuit component development, system processor and software architecture require exploration. Since an exotic multiprocessor architecture is sought, then a rule-based SAG is employed to configure alternative processor interconnection schemes according to locally developed hueristics. Each processing element corresponds to a cell in the processor interconnection spreadsheet. Simulations of intercell message traffic and processing load can now be performed. In order to get better simulation results, cell processors can actually execute prototypical software modules developed in parallel to the hardware and circuit design. Further, in order to achieve adequate simulation turnaround and throughput, cell processes are migrated from an initial single processor, to a network of processors that now use the spreadsheet as a message routing and analysis blackboard interface.

We must also address how to organize the staff of system development specialists. Based upon our prior studies of CAD-VLSI work [51], we know that problems of system consistency and integration can emerge in the development organization. These problems give managers headaches. System components whose development falls behind schedule may have logic flow or timing errors since staff must work under increasing pressure and competing demands for their attention. Tools and techniques that support circuit, software, and system configuration management must be put in place, monitored, and coordinated. The complexity of this task suggests that a knowledge-based project planning, scheduling, and management system be employed to aid in project coordination. However, project staff must be committed to routine use of the PMSS during development. Therefore, project managers and staff are provided regular access to this project management support system, and can query/update project milestones, relative progress, and potential development problems.

Thus, the SF can be specialized to support the parallel development of a multiprocessor system employing new circuit and prototypical software components, all in a manner that accoodates an integrated organizational approach.

6. Experiences in SF Technology Transfer

Some people hold that the primary purpose of a school of science, engineering, or information technology is technology transfer. Such transfer nominally occurs via educated students who take their newly acquired academic scholarship to their workplace, and then apply this knowledge to practical problems. We consider this the baseline for software technology transfer. But the successful practice of the software technology transfer (STT) on a larger scale requires understanding much more than this [48]. Subsequently, we use this section to describe some of our experiences in STT.

We observe two different modes of STT through the SF project: *intraorganizational* and *interorganizational*. Within the SF project, the value of internal STT is taught and practiced. This happens in three ways. First, project teams are given access to the prior year team project deliverables. Students are encouraged to reuse and incorporate the prior project life cycle products (e.g., subsystem functional specifications and source implementation) into their own, with shared products duly acknowledged. Many project teams choose to do this and therefore are able to deliver more substantial and more capable products. Second, since project teams are focussed primarily on their respective tools or subsystems to engineer, all team members are required to conduct project specification and design reviews for some other project component. This gives the student staff an opportunity to evaluate and compare their deliverables, and provide orchestrated quality assurance reviews. This also gives the students a chance to learn more about other parts of SF products and processes. Third, students are directed to use different SEE components to support the development of their deliverables. Thus, students become users of advanced prototype software engineering tools, and can evaluate the tools'

strengths and weaknesses when applied to in a large project setting.

Moving SF products, production techniques, or project management strategies into other organizational settings takes the preceding STT efforts further. Interorganizational STT occurs in a number of ways. First, all students who complete their project deliverables are encouraged to share them with other team members and to take to their workplace.[17] Thus, students can take parts of the SF away with them to their workplace.

Second, some entreprenuerial students have elected to use their project deliverables as the basis for developing commercial software products. They seem confident in their own technical ability and in the capabilities of the prototypes products they have delivered as part of their coursework. The personal computer software arena seems to be a favorite area these budding entreprenuers seek to find a niche for their eventual product. However, the emerging computer-aided software engineering (CASE) marketplace is gaining more attention. Many of the student start-up firms remain relatively small and last less than two years. On the other hand, others seem to survive and grow. But such experiences are part of process of real-world continuing education in STT for these students.

Third, as witnessed through this report, researchers participating in the SF project (such as the author) regularly produce research publications, conference presentations, and academic/industrial research colloquia for peer group consumption. A growing number of research publications are co-authored by students writing about the products of their completed coursework in the SF. Students can thus pursue the option of diffusing their knowledge and technological concepts through research publications, thereby adding value to both the educational experiences and professional status.

Next, as the SF project continues to push both the state of the practice and the state of the art in LSSE, various industrial organizations actively seek access to SF products, production processes, and project management strategies. These technologies are diffused through consulting contracts, contract research, and licensing arrangements. As might be expected, consulting arrangements facilitate the transfer of expertise through on-site short courses, project or proposal reviews, and product/process designs via research reports or technical memoranda. Contract research represents a greater level of commitment from an industrial firm, usually in the form of an IR+D subcontract. We restrict our involvement to undertakings directed at basic software engineering research problems rather than commercial products. Accordingly, we deliver our research results in the forms of technical reports or prototype systems. For example, for one contractor, we delivered a prototype system specification generator that accepts structured English requirements and paraphrases them into an operational specification language. In another case, we are actively exploring advanced techniques for specifying and documenting distributed, real-time multi-processing systems. Since we use the SF project to produce these technologies, we also then incorporate these products into subsequent SF configurations. Last, technology licensing agreements represent the direct acquisition of SF deliverables through the university for commercial purposes. Here we find interest not only in acquiring prototype software implementations, but also (or sometimes only) interest in acquiring rights to software specifications or designs, since these are viewed as being technologies reusable in many potential applications.

Clearly, all of the the preceding channels for diffusing SF technologies into other organizational settings do not complete the STT process [48]. The successful transition and prolonged use of the concepts, artifacts, and packages we produce may take years. However, we do note that most of our industrial sponsors seek sustained relations that get reiterated, and we continue to be approached by more firms interested in gaining access to SF technology and student staff.

---

[17]Such exchanges require an agreement not to further disseminate, license, or market other people's work or university resources without prior written agreement. Thus, the agreement is intended to have the form of an extended loan of borrowed property, but not an entitled claim to ownership or exclusivity.

7. Conclusions

Developing the next generation of large-scale software systems will change the patterns of work of system development organizations. We therefore expect the major system engineering problems to be solved will require organizational solutions that accomodate advanced software development tools, flexible manufacturing techniques for system life cycle engineering, and strategies for managing large system development projects. Through the System Factory project, we have created an experimental organizational environment for developing large software systems that allows us to encounter these problems and find effective solutions. The SF approach also gives us a base of experience for considering how it might be applied in the future.

The SF emerged over a seven year development effort. This effort produced a series of successive environments that served as prototypes for subsequent versions, see [39, 20, 41, 44, 12] and above. Some SF CASEE components such as the language-directed editor generator were developed through five complete development life cycles, while others such as the knowledge-based project management support system have passed through fewer prototyping cycles. As such, the SF CASEE is conceived to exploit common reusable software tool components, cyclic prototyping, and multi-stage development techniques. Similarly, the SF software life cycle engineering techniques and project management strategies have been prototyped, put into practice, and refined. This approach to *continually emerging system development* also leads us to value redundant functionality across different SF technologies. This allowed us to undertake LSS development efforts that would inevitably encounter outstanding problems in LSSE project organization, but with opportunities for their resolution. As such, it is an effective approach to improve total staff productivity and skill level, as well as provide alternative means for achieving desired LSSE products or services.

The System Factory approach to LSSE is directed to exploring alternative organizational forms of software manufacture and evolution. In our view, software manufacture should be set up, staffed, and managed as an organizational unit that merges the flow of production resources (e.g., reusable software component families) through a large-scale production process and software development infrastructure. In turn, these organizational technologies should structure the work of software developers to produce software application systems, descriptions, and environments in a productive, cost-effective, and high-quality manner. Subsequently, we speculate that if this research is continues to be successful, it should be possible to specify or develop turn-key system factories for other high technology applications as another product of our research.

The SF project provides a hands-on learning, research and development experience in LSSE for all project participants. We have demonstrated the large-scale software engineering project coursework can prototype complex full-scale systems in a relatively short time. Our early experiments in developing a computer-aided software engineering environment demonstrate this. Similarly, we were able to integrate and operationalize a diverse collection of reusable software tools and life cycle engineering techniques. A LSSE effort such as the SF project immediately gives project participants insights into the importance identifying and practicing many different project management strategies. This happens as soon as participants find that the most difficult LSSE problems to solve are primarily organizational, rather than technical. Last, although seemingly ignored in most software engineering coursework, we find that software technology transfer can be taught and practiced by all LSS project participants. As such, we believe that other software engineering researchers and educators should consider adopting reiterated and sustained project courses such as the SF project.

Developing LSSE projects is possible but by no means limited to the types of tools, packages, or software engineering environments developed in the SF. For example, domains amenable to LSS development include graphic programming environments, production of "feature-length" computer animated movies, reconfigurable user interface management systems, group design of application-specific VLSI processors, interactive CD ROM-based undergraduate CS curriculum courseware, system factories for computer integrated manufacturing, environments and simulators for developing integrated multi-sensory intelligent systems, and so forth.

Many universities are becoming equipt to conduct LSS development projects due to extensive

institutional commitments to create computerized campuses [56]. Most of these universities will seek a heterogeneous, open system network of computing resources distributed across many institutional locations. This increasing diversity of computing environments will lead to a greater spanning of multiple organizational units in order to successful complete LSSE projects. In our view, the trailblazers best positioned to capitalize on such opportunities will be faculty and researchers in the area of large-scale software engineering.

8. References

1. R. Balzer, N. Goldman, and D. Wile. "Operational Specifications as the Basis for Rapid Prototyping". *ACM Software Engineering Notes 7*, 5 (1982), 3-16.

2. R. Balzer. "A 15 Year Perspective on Automatic Programming". *IEEE Trans. Software Engineering SE-11*, 11 (1985), 1257-1267.

3. S. Bendifallah and W. Scacchi. "Understanding Software Maintenance Work". *IEEE Trans. Software Engineering 13*, 3 (1987), 311-323.

4. P. Bernstein. *Workplace Democraticization: Its Internal Dynamics.* Kent State University Press, 1976.

5. B. Boehm. "Seven Principles for Software Engineering". *J. Software and Systems 3*, 1 (1983), 3-24.

6. J.S. Brown and S. Newman. "Issues in Cognitive and Social Ergonomics: From Our House to Bauhaus". *J. Human-Computer Interaction 1*, 4 (1986).

7. J. Burbridge. *The Introduction of Group Technology.* John Wiley & Sons, 1975.

8. A. Castillo, S. Corcoran, and W. Scacchi. A Unix-based Gist Specification Processor: The System Factory Experience. Proc. 2nd. Intern. Conf. Data Engineering, 1986, pp. 582-589.

9. D. Cohen. Symbolic Execution of the Gist Specification Language. Proceedings IJCAI-83, 1983, pp. 17-20.

10. R. Cronk and D. Zelinski. ES/AG: A System Generation Environment for Intelligent Application Software. Proc. SOFTFAIR II, IEEE Computer Society, 1985, pp. 96-100.

11. J. Distaso. "Software Management - A Survey of Practice in 1980". *Proceedings IEEE 68*, 9 (1980), 1103-1119.

12. L. Eliot and W. Scacchi. "Toward a Knowledge-Based System Factory: Issues and Implementations". *IEEE Expert 1*, 4 (1986), 51-58.

13. S. Feiner, D. Salesin, and T. Banchoff. "Dial: A Diagrammatic Animation Language". *IEEE Computer Graphics and Applications 2*, 7 (1982), 43-56.

14. M.S. Fox and S.F. Smith. "ISIS: A Knowledge-based System for Factory Scheduling". *Expert Systems 1*, 1 (1984), 25-49.

15. P.K. Garg and W. Scacchi. "A Hypertext System to Manage Software Life Cycle Documents". *IEEE Software 5* (1988), (to appear).

16. L. Gasser. "The Integration of Computing and Routine Work". *ACM Trans. Office Info. Sys. 4*, 3 (1986), 205-225.

17. E.M. Gerson and S.L. Star. "Analyzing Due Process in the Workplace". *ACM Trans. Office Info. Sys. 4*, 3 (1986), 257-270.

18. B. Hannaford. The Electronic Spreadsheet: A Workstation Front-End for Parallel Processors. Proc. COMPCON 1986, 1986, pp. 316-321.

19. C. Hewitt. "The Challenge of Open Systems". *BYTE 10*, 4 (1985), 223-242.

20. R. Katz, W. Scacchi, and P. Subrahmanyam. "Development Environments for VLSI and Software Engineering". *J. Sys. Soft. 4*, 2 (1984), 14-27.

21. B.I. Kedzierski. Knowledge-Based Project Management and Communication Support in a System Development Environment. Proc. 4th. Jerusalem Conf. Info. Techology, 1984, pp. 444-451.

22. T. Kidder. *The Soul of a New Machine.* Avon Books, New York, 1982.

23. R. Kling and W. Scacchi. "The DoD Common Higher Order Programming Language Effort (Ada): What Will The Impacts Be?". *SIGPLAN Notices 14*, 2 (1979), 29-41.

24. R. Kling and W. Scacchi. Recurrent Dilemmas of Computer Use in Complex Organizations. Proc. 1979 National Computer Conference, 1979, pp. 1067-116. Vol. 48.

25. R. Kling and W. Scacchi. "Computing as Social Action: The Social Dynamics of Computing in Complex Organizations". *Advances in Computers 19* (1980), 249-327. Academic Press, New York.

26. R. Kling and W. Scacchi. "The Web of Computing: Computer Technology as Social Organization". *Advances in Computers 21* (1982), 1-90. Academic Press, New York.

27. D. Lenat, A. Borning, D. McDonald, C. Taylor, and S. Weyer. Knoesphere: Building Expert Systems with Encyclopedic Knowledge. Proc. IJCAI-83, 1983, pp. 167-169.

28. N. Magnenat-Thalmann, D. Thalmann, and M. Fortin. "Miranim: An Extensible Director-Oriented System for the Animation of Realistic Images". *IEEE Comptuer Graphics and Applications 5*, 2 (1985), 48-73.

29. (multiple authors). "The VHSIC Hardware Description Language". *IEEE Design and Test 3*, 2 (1986), 13-57.

30. K. Narayanaswamy and W. Scacchi. "A Database Foundation to Support Software System Evolution". *J. Systems and Software 7* (1987), 37-49.

31. K. Narayanaswamy and W. Scacchi. An Environment for the Development and Maintenance of Large Software Systems. Proc. SOFTFAIR II, 1985, pp. 11-25.

32. K. Narayanaswamy and W. Scacchi. "Maintaining Configurations of Evolving Software Systems". *IEEE Trans. Soft. Engr. 13*, 3 (1987), 324-334.

33. R. Prieto-Diaz and J. Neighbors. "Module Interconnection Languages". *J. Sys. and Soft. 6*, 4 (1986), 307-334.

34. P.G. Ranky. *The Design and Operation of FMS: Flexible Manufacturing Systems.* North-Holland, 1983.

35. U. Rembold and R. Dillman (eds.). *Methods and Tools for Computer Integrated Manafacturing.* Springer-Verlag, 1984. Lecture Notes in Computer Science, Vol. 168.

36. C. Reynolds. "Computer Animation with Scripts and Actors". *Computer Graphics 16*, 3 (1982), 289-296.

37. A. Sathi, T. Morton, and S. Roth. "Callisto: An Intelligent Project Management System". *AI Magazine 7*, 5 (1986), 34-52.

38. W. Scacchi. *The Process of Innovation in Computing: A Study of the Social Dynamics of Computing.* Ph.D. Th., Department of Information and Computer Science, University of California, Irvine, 1981.

39. W. Scacchi. A Language-Independent Software Engineering Environment. Workshop Report: VLSI and Software Engineering, IEEE Computer Society, 1982, pp. 99-103. IEEE Catalog No. 82CH1815-0.

40. W. Scacchi. Developing VLSI Systems with a Silicon Engineering Environment. IEEE Intern. Conf. on Computer Design, IEEE, 1983, pp. 472-475.

41. W. Scacchi. The System Factory Approach to VLSI and Software Engineering. Proceedings AFCET Second Soft. Engr. Conf., 1984, pp. 349-359.

42. W. Scacchi. "Managing Software Engineering Projects: A Social Analysis". *IEEE Trans. Soft. Engr. SE-10*, 1 (January 1984), 49-59.

43. W. Scacchi. Applying Social Analysis of Computing to System Development. Proceedings Aarhus Conference on Development and Use of Systems and Tools, August, 1985, pp. 477-500. Aarhus, Denmark.

44. W. Scacchi. The Software Engineering Environment for the System Factory Project. Proc. 19th. Hawaii Intern. Conf. Systems Sciences, 1986, pp. 822-831.

45. W. Scacchi. "Shaping Software Behemoths". *UNIX Review 4*, 10 (1986), 46-55.

46. W. Scacchi. Software Specification Engineering: An Approach to the Construction of Evolving Software Descriptions. USC/Information Sciences Institute, 1987. (in preparation).

47. W. Scacchi. The System Factory Approach to Software Engineering Education. Educational Issues in Software Engineering, 1987.

48. W. Scacchi and J. Babcock. Understanding Software Technology Transfer. internal report, Software Technology Program, Microelectronics and Computer Technology Corp., Austin, TX.

49. W. Scacchi and C.M.K. Kintala. Understanding Software Productivity. technical memorandum, Advanced Software Concepts Group, ATT Bell Laboratories, Murray Hill, NJ.

50. W. Scacchi, S. Bendifallah, A. Bloch, S. Choi, P. Garg, A. Jazzar, A. Safavi, J. Skeer, and M. Turner. Modeling System Development Work: A Knowledge-Based Approach. Computer Science Dept., USC, 1986. working paper SF-86-05.

51. W. Scacchi, L. Gasser, and E. Gerson. Problems and Strategies for Organizing Computer-Aided Design Work. Proceedings IEEE Intern. Conf. Computer-Aided Design, 1983, pp. 149-152.

52. T. Schwederski and H.J. Siegel. "Adaptable Software for Supercomputers". *Computer 19*, 2 (1986), 40-48.

53. S. Sluizer and P. Cashman. XCP: An Experimental Tool for Managing Cooperative Activity. Proc. 1985 ACM Computer Science Conf., 1985, pp. 251-258.

54. W. Swartout. The Gist Behaviour Explainer. Proceedings AAAI-83, 1983, pp. 402-407.

55. R. Thayer, A. Pyster, and R. Wood. "Major Issues in Software Engineering Project Management". *IEEE Trans. Software Engineering SE-7*, 4 (1981).

56. D.A. Updegrove. "Computing Intensive Campuses: Strategies, Plans, Implications". *EDUCOM Bulletin 21*, 1 (1986), 11-14.

57. R. Waters. "The Programmers Apprentice: A Session with KBEmacs". *IEEE Trans. Software Engineering SE-11*, 11 (1985), 1296-1320.

58. S. Weyer and A. Borning. "A Prototype Electronic Encyclopedia". *ACM Trans. Office Info. Sys. 3*, 1 (1985), 63-88.