The SIGMA experience—A study in the evolutionary design of a large software system

by DAVID WILCZYNSKI and RONALD TUGENDER

USC/Information Sciences Institute
Marina del Rey, California

and

DONALD OEESTREICHER

Xerox Corporation
El Segundo, California

INTRODUCTION

Anyone who has been a part of a large software effort knows of its peculiar afflictions and special problems. The literature teems with guidelines, rules and conventions for designing and managing such systems; in fact, there are probably more books written about large systems than there are large systems. This paper will make no attempt to add to this literature; instead it will simply report our experience over the last five years in the design and implementation of SIGMA. This introspective study is meant to be not a pedagogical paper but a reflective, often humbling, diary.

SIGMA is the interactive message processing system built at the University of Southern California Information Sciences Institute (ISI) for the Military Message Experiment (MME). It is currently in active use at Camp Smith in Hawaii, supporting about 100 users, of which 24 may be concurrently online. SIGMA comprises about 270 modules totaling some 2500 routines. This makes approximately 200,000 lines of source code or some two edge-feet of listings. The point, of course, is that SIGMA is big by any metric. The software was designed and written by a revolving group of five to seven expert programmers, making SIGMA about a 30 man-year program. This is by no means a gigantic effort by industrial standards, but large when judged by research community criteria.

Our experience should not be taken lightly. Deadline pressures notwithstanding, the ISI environment is conducive to high-quality output. Each project member has a private office, an HP2640A CRT terminal, access to an inhouse library and all the facilities one expects at a computer research center. Our successes or failures are generally a result of our own ingenuity and intelligence or lack thereof, respectively.

SIGMA ARCHITECTURE OVERVIEW

The SIGMA message service runs on a Digital Equipment PDP-10 under the TENEX timesharing system and is written in the BLISS system implementation language. The SIGMA system is divided into two functional areas: the user jobs, which interact with the message service users; and the daemons, a collection of background processors that perform non-interactive functions.

The SIGMA user job

An instance of the SIGMA user job is created for each user who logs into the system. As seen in Figure 1, the user job is composed of five major components. The Terminal Driver interfaces the specially modified HP2640A terminal to the rest of the user job. The Command Language Processor (CLP) reads command input, parses it, builds a command specification called an Execution Request Block (ERB), and passes it to the Functional Module (FM), through a protocol called EC99. The FM is responsible for the actual execution of commands, and thus it has two main tasks: to manage the display of information on the terminal, and to manipulate SIGMA's objects. The former task is performed by a module called the Virtual Terminal (VT), which builds and maintains display lists for the terminal. The latter function is done by the FM directly for text objects and selectors, and by two

---

* SIGMA supports four kinds of objects. Text Objects are lists of uninterpreted paragraphs. Messages, of which there are several kinds, are lists of various message fields. Folders are lists of message citations; a citation is an abstract of its associated message. Selectors are boolean expressions applied to folders in order to access only those citations whose attributes match those named by the selector.
special-purpose modules called the Folder Module (FACMOD) and Message Module (MSGMOD) for folders and messages, respectively. Because of address space limitations, FACMOD and MSGMOD are located in separate TENEX forks (processes) and require a special Inter-Fork Communication Protocol (IFCP) to communicate with the FM.

The SIGMA daemons

The SIGMA daemons are responsible for tasks that require no direct user interaction and can thus be performed in background. These include management of the shared data bases (messages and folders), message reception and transmission, archive retrieval, and printer spooling. The daemons process requests received through input queues; the source of such requests may be the user jobs or other daemons. To provide operations personnel with the ability to control the daemons, a Configuration Control Program (CCP) communicates operator requests to daemons.

A SIGMA TIMELINE

Figure 2 represents our first effort at producing a milestone diagram for SIGMA's development. This event-oriented chart provides a general impression of how SIGMA developed.

The project, which started in September 1973, was first called Information Automation (IA). Approximately one year later a series of six IA papers were produced defining the components of what was to be SIGMA and by January 1975 a full system design was published. This 200-page document represented about ten man-years of effort culminating the period we now call "designing in a vacuum."

The first SIGMA, which we will call SIGMA 0 (although it was both unnumbered and unnamed), made its debut a year later in December 1975. It didn't do much, but it did have a front-end that talked to users, a minimal FM that executed a few message manipulation commands, a terminal simulator that was to be functionally equivalent to the terminal being designed for our project, a primitive debugging mechanism, and a simple text editor.

The next year was spent getting SIGMA 1.0 ready for a system evaluation which was to take place in Boston in February 1977. This pubescent period saw SIGMA take on functional substance. The first daemons were written; the concepts of folders, text objects, and selectors were implemented; and the FM was rewritten to encompass all the new objects. SIGMA 1.0 was slow and bulky, but showed enough promise to be the service selected for the Military Message Experiment. We knew even then that the daemon design and implementation were hopeless, but other things had to be done first. SIGMA's performance had to be improved.

Performance was the project watchword for SIGMA 1.75. Since the functionality had become stable, we were able to carefully analyze the system's flow of control, data paths, communication demands and so forth, a study which led to
optimizations spanning all parts of the user job. This performance transition was particularly trying. Still, by December 1977 it was running at an acceptable speed. The changes we made were so dramatic that SIGMA 1.0 seemed like ancient history.

The time had finally come to deal with the daemons. Their original design was based on principles that led to an overly-complicated implementation whose data flow can euphemistically be described as baroque. The irony is that by SIGMA 1.75 we realized that all the requirements that led to this design were either too expensive or superfluous. This time the daemons were redesigned and rewritten based on our experience of what background support was actually needed.

The new daemons had a great effect on our project. The 1.75 daemons were hopeless to maintain, difficult to debug, and resistant to change; the 2.0 daemons of July 1978 were elegant by every programming standard. SIGMA could finally be called a mature system. Now and only now could we think about functional enhancements that involved the daemons.

The remaining part of 1978 was spent preparing for SIGMA 2.2. Among the new features was Alert processing, an addition which made the online user immediately aware of activity in his personal pending file. The Alert concept had been knowingly lacking since SIGMA 1.0 but was not practical to implement until we redesigned the daemons and had a stable user job.

While in this timeline we seemed to be responding to individual and isolated problems as they appeared, a more abstract view of the development reveals a coherent top-down design. The first year we designed a message service in great detail. The first SIGMA was produced in year two with the emphasis on the front-end components, the CLP and the terminal. The functionality, defined and implemented during year three for SIGMA 1.0, was demonstrated in Boston. Following that came the architectural wirebrushing of the user job, culminating in SIGMA 1.75 in year four. The new functional daemons came during the fifth year together with a round of improvements for SIGMA 2.2.

Even though we actually followed a sensible path in SIGMA's development, our failure to realize what was happening proved to be both costly and frustrating. No doubt all retrospective analysis is concise and relevant; still, early recognition of certain principles would have prevented the chaotic nature of our progress. The next section will present those principles as a pseudo-mathematical theory of software development.

LARGE SYSTEM DESIGN THEORY

Now that SIGMA has matured we compared its current design and implementation with the one planned in the initial system design. This comparison shows that about the only thing that survived all those years is the system diagram found in Figure 1. There is something futile about the first axiom:

Axiom 1—Early system design should identify only large namable components. Perhaps this is the best that you can do when designing in a vacuum. SIGMA 0 wasn't very good but it did show that our top-level design was sound. Lacking some sharp insight we believe that:

Axiom 2—A complete running system is necessary to verify even a top-level design.

The implications of this axiom are severe. When the system is being put together, a placeholder is needed for every component regardless of expense. We knew, for example, that even though the SIGMA terminal was scheduled for late 1976, we needed a simulator for it in order to design the CLP and FM. The man-years spent on this known throwaway piece of software were eminently worthwhile, substantiating the following:

Theorem 1—Every major component of a design must be implemented in some form.

Our FM in SIGMA 0 produced the necessary results. It showed that the CLP/FM interface was sound and the terminal model was maintainable by the VT. Even though the
FM barely worked, nothing would have been learned if it didn't run. Trite as it may sound:

Corollary 1.1—All implemented components must work.

SIGMA 0 showed us that some of our high-level concepts were designed correctly. However, the implementation experience also taught us other things.

The original FM had too many features, among them a concept called Placemarks. These were named locations within message bodies that the user could assign and address for various purposes. They were hard to implement, and in fact did not survive the transition to SIGMA 1.0. Placemarks for various purposes. They were hard to implement, and in fact did not survive the transition to SIGMA 1.0. Placemarks may or may not have proved useful, but this was not the time to find out; suffice it to say that much valuable time was wasted here.

ACCMOD was the first complete message manager for SIGMA. It had a data base model for the way it serviced the FM. In other words, it owned the message being displayed and was involved in every facet of its editing. Besides just having performance problems, ACCMOD was a huge, unwieldy piece of code. In the simpler file model, MSGMOD gets the message to the FM who "owns" it during the editing session, and then MSGMOD stores it when the user is done. The important knowledge we obtained from ACCMOD was that its model was wrong. However, a simpler implementation would have shown this as well.

One thing we designed right was the communication of common data between the forks of the user jobs. Data (such as who was logged on, the ID of the open message, etc.) was kept in memory shared between them. The effect was like a FORTRAN COMMON block. In the simpler file model, MSGMOD gets the message to the FM who "owns" it during the editing session, and then MSGMOD stores it when the user is done. The important knowledge we obtained from ACCMOD was that its model was wrong. However, a simpler implementation would have shown this as well.

The Placemark/ACCMOD/Common experience shows clearly that a project should:

Theorem 2—Start small in early design and implementation phases.

It is hard to overestimate the pragmatic value of Theorem 2. Its realization leads to a natural evolution in which the designers can cheaply reflect on their basic ideas and perhaps modify them before they run out of funds or energy. We should note that there were no daemons at this time.

We were still designing them as if they would suddenly appear like Pallas Athena, springing full-grown from the forehead of Zeus. An understanding of Theorem 2 would have led us to a more conservative first-pass design.

The work which led to SIGMA 1.0 represented a new direction for the project, one that was not anticipated at the time. SIGMA 0 verified our front-end model; SIGMA 1.0 would define the system's functionality as needed by the FM. Had we realized that this was the real effort, we could have avoided the complexity found in the daemons. In other words:

Axiom 3—You can't aim high on every component at the same time.

Many things happen during a design cycle: Ideas are tested, system modules are inspected, some things work well, and some things are thrown out. Typically, some things stabilize. The CLP did by the time SIGMA 1.0 was released.

The FM was beginning to at the same time. Unfortunately, what had stabilized throughout the user job was our text-handling routines called ZT. ZT was a loose collection of low-level functions. Given these primitive routines each implementer developed his own set of macros and functions, all based on the ZT structure. By the time we realized its performance and design implications, ZT was hard to remove from the user job. The TEXT package that replaced ZT simplified SIGMA by imposing a coherent text-processing model on the project. What we should have realized is:

Axiom 4—Recognize which component is stabilizing during a design cycle and aim high for it.

These two axioms are the heart of our evolutionary design theory. Even with unlimited resources, we believe a project will do better to focus its attention on one major component at a time. Some scientific reasons, ranging from interface issues to manpower expenditure, are easy to generate. But one which is often missed is that very little is learned when an overdesigned module is rewritten or modified—it is better to build up from minimum capabilities than tear down from unneeded ones. This "hourglass design," in which one starts high and must then strip away features before a redesign is possible, is a painful way to develop a system.

Consider the daemons. Their original design was based on four requirements:

1. SIGMA would run in a distributed network environment.
2. High-priority requests would need to be processed quickly.
3. Individual long requests could not be allowed to hold up the daemons.
4. Error results must be returned to the user in a synchronous mode.

The ramifications of these early requirements were severe. The first implied a logon procedure so that the daemons knew the location of its users. This meant that when the daemons went down so did all the users, bad in operational use in Hawaii and hopeless for development at ISI. The second requirement led us to an implementation of duplicate processors with different priorities in case a high-priority request came in. The "multiprocessor" environment was also prepared for the third requirement in which a request could monopolize a processor. Finally, returning requests to the user, as necessitated by (4), implied a single output process for distributing results.

By SIGMA 1.75 all those requirements were shown to be spurious and the code supporting them was removed. What was left were daemons with the barest capabilities. From them together with our experience with 1.75 we were able to redesign and rewrite the daemons to be elegant and functionally relevant in just four months. This lesson is our first major result:

Theorem 3—Avoid the hourglass design syndrome.

Figure 3 depicts this theorem graphically. The syndrome is seductive and debilitating. The natural ego of a designer/programmer drives him to build to the hilt. We've all seen this happen in design meetings. A few capabilities are clearly required; some others are suggested and incorporated. Once
rolling, this juggernaut is hard to stop, and a concise design suddenly belongs to a committee. PL/I is an old example; SIGMA’s former daemons are our contribution.

FACMOD is an example of how smoothly evolution can take place. Folders contain message entries and are the place from which messages are displayed. Besides just being displayable, entries can be keyworded, deleted, filed into other folders, commented, selected by various criteria, etc. Instead of building in all these capabilities, the first FACMOD had only the barest set. FACMOD came up quickly and grew with our needs. Naturally, many of the requirements we foresaw were added later, but many were not.

The message is clear:

Theorem 4—Underdesigned components are needed during a system’s evolution.

We have talked about system design in terms of focus on mainline modules, aiming high in selected pieces, underdesigning others. Axiom 4 also alluded to stabilizing factors within all parts of the system. Those are important to watch for. As important are sections that are critical for one reason or another. Two examples in SIGMA are the IFCP package and the Terminal/Terminal Driver communication protocol.

The IFCP is the data channel between the FM and both MSGMOD and FACMOD. It was independent of application and unlikely to change regardless of the direction of any of those components. The IFCP had to be robust and solid; it was serving many masters. It was recognized as a critical component of SIGMA and designed accordingly.

The terminal protocol raised different issues. Even though the transmission lines at ISI between the terminal and the Driver were virtually perfect, we knew that lines at other installations might not be. Without knowing how bad lines could be, we anticipated the worst and built a robust protocol that included acknowledgments, timeouts, retransmissions, and checksums. Our experience in the Boston review and in Hawaii made us glad that we did. So, underdesign is good, but:

Theorem 5—Aim high for critical sections.

Part of aiming high is to understand in detail what is expected of the component under examination. Before the daemons were redesigned in aim-high mode, a firm requirements specification was written for them. This document gave our design and implementation a well-defined target. The VT never has received this treatment and remains loosely organized. It seems that part of aiming high means that:

Corollary 5.1—Firm requirements are needed for mainline components.

Requirements are one thing, and documentation is another; the former precedes implementation while the latter follows it. Once a major component of the system has been written, complete documentation should be generated for it. Much lip service is paid to this liturgy; we are no exception. All the good reasons for documentation are easy to list but one that is understated is that undocumented components take on an orphan flavor if the writer leaves. This happened to the VT. It passed from hand to hand, each programmer leaving his mark, but no one documenting anything. So:

Corollary 5.2—Fully document mainline components as they are written.

As important as Theorem 5 and its corollaries are, their duals are just as important:

Theorem 6—Aim low for noncritical system components.

Corollary 6.1—Noncritical components need only tentative requirements.

Corollary 6.2—Small documentation effort should be spent on noncritical components.

Noncritical system components should gain aim low treatment—the daemons and ACCMOD should have but didn’t, FACMOD and COMMON should have and did, the VT and ZT shouldn’t have but did. The technical issues here are not controversial. However, the psychological one of producing work that is less than your best arises. Perhaps a good manager should assign his best programmer to produce such code, since ego is less likely to be a problem. Even though the code is known to be throw-away, it still has to work (Corollary 1.1).

The theme of aim-low also brings questions of performance. The development leading to SIGMA 1.0 paid virtually no attention to performance. We used flexible data structures, clean interfaces between modules, and straightforward coding techniques. SIGMA 1.0’s performance was poor but not unexpectedly so. When performance became an issue, we analyzed SIGMA using Program Counter (PC) samples, a technique that takes snapshots of running code to tell what code is being executed. We found, for example, that our storage management package needed tight optimization, since it was active all the time. That was expected, but we also found that some heavily used operating system facilities took several orders of magnitude more time than we expected. This stunning information taught us that:

Axiom 5—A priori, it is impossible to know in which sections of code performance will be critical.

With the PC samples as a guide optimizing SIGMA’s code in the user job was straightforward: the character output to the terminal was rewritten at a 10-to-1 saving, the FM’s folder handling was changed from linked lists to an array structure, FACMOD and MSGMOD calls were tuned to specific needs, and a TEXT package was incorporated to replace ZT. These changes were extensive and revolution-
Theorem 7—Performance should not be an early design goal.

This theorem promotes an ideal which served SIGMA poorly. In following it we found ourselves with a SIGMA 1.0 that was very slow when compared to one of our competitors during the Boston review. The "fast" system did not have the capabilities of SIGMA, although that fact together with the above performance principles was lost on some (fortunately not all) of the reviewers. The performance of SIGMA 1.75 vindicated our development strategy.

Until now the discussion has addressed design theory with an undercurrent of implementation examples. With the evolutionary principles we are proposing, design and implementation are inextricably tied together. Now the focus will shift to the implementation side.

The literature rightly pays much attention to storage handling, searching and sorting, queue modeling, etc., because of their widespread use. To have poorly implemented packages of this sort is ludicrous. whether you are aiming high or low. Besides avoiding code duplication, these modules have a unifying effect on a system. Our careful implementation of free storage, lexicons, and queues were based on the principle that:

Axiom 7—Good support packages are essential and well understood.

Support tools are important, but a good supporting environment for testing is essential. From the beginning we realized that since BLISS had no debugging facilities we would have to provide our own. A system-error package was built to take over control when an error was detected. The programmers had an ASSUME (boolean,message,data) statement available that invoked the system-error mechanism if the boolean was false. The package was small when we started (no hourglass design here) and grew as SIGMA evolved. Note however, that small and aim-low are not the same:

Theorem 8—Aim high on debugging tools.

It is time to formalize an assumption that conveys our theory. We have been cavalier about the implementation effort required by our evolutionary model. Yet each design cycle, whether it was the CLP, the FM or the daemons, was followed by a painless implementation phase. Once understood:

Theorem 9—System components are easy to build with good support tools.

This optimistic theorem runs contrary to prevailing, perhaps self-serving, opinion. Our contention that well defined modules can be cranked out comes from innumerable cases, from both SIGMA and other sources. It's just not hard to build "one." Yet what we have tried to show throughout this paper is that a large system needs to evolve, since many decisions can be wrong. We have shown how things can be designed or modeled poorly.

Even knowing where to put functionality can be a problem. As an example, the original SIGMA design included a component called the Personal Daemon (PD). An instance of this background process was created for each user job, and its intended purpose was to provide a background processing capability which was active even when the user was not. Alerts were originally designed to be a PD function. Since we neither knew all the potential functions of such a process nor had the time to develop them, we (correctly) gave it aim-low treatment. As the design of SIGMA was reworked from 1.0 to 2.2, each of the existing hypothesized PD functions was reassigned to another area. With nothing left to do, the PD was removed. Unfortunately:

Axiom 8—Often you guess wrong.

Every component of SIGMA has been rewritten. Large pieces of code were abandoned: ACCMOD (twice), the FM, the daemons, the PD. It is the nature of the evolutionary design theory that:

Theorem 10—Almost everything gets thrown away.

Even aim-high modules can be re-examined and thrown away. SIGMA had a directory scheme for storing and retrieving messages based on a lexicon package (height-balanced AVL trees). It worked from the beginning and never was looked at until our performance cycle. When it got dumped for a simpler scheme we knew nothing was sacred:

Corollary 10.1—Don't be afraid to throw stuff away.

Once this fear is conquered, a lot of pressure is removed from a design/programmer staff—certain kludges are accepted, simplicity encouraged. The aim-low paradigm coupled with Theorem 10 means that each component will get a second chance during an aim-high cycle. The CLP got it by SIGMA 0. Functionality made the FM the focus of SIGMA 1.0. The entire user job got aim-high treatment for 1.75. Finally, the daemons were the target for 2.0. Each cycle left a little more of SIGMA hardened, i.e.,

Corollary 10.2—Every development cycle will focus on a permanent component.

Even though we didn't realize it at the time, SIGMA followed an orderly development cycle. It would be nice to attribute this outcome to our careful long-range planning, but that is not the case. The things we did were circumstantially good or bad. Performance, redesign, and new requirements all imply that a large system is a moving target. Simply stated, the first fundamental result is:

Theorem 11—No one can implement a large system in one pass.

It is seductive to think that you can. Even more seductive is the notion that a long, studious design is the answer; we tried and failed. Every aspect of SIGMA proved the statement made by our last theorem:

Theorem 12—A large mature system must evolve; it cannot be designed.

This second fundamental result is borne out by all large
systems. There is no good reason to suspect that anyone can design a big system down to the bit level using any known methodology.

CONCLUSIONS

The theme of this paper is applicable only to large, multi-year projects. Though many of the principles are relevant to any effort, the theory applies to a "design a little, implement a little, design some more, implement some more . . ." paradigm, rather than the "design it, implement it" one. The most important early decision is to recognize which model is appropriate. Remember the two fundamental theorems.

A relevant issue, not previously mentioned, is the choice of a programming language for a project of SIGMA's magnitude. When we faced this decision, INTERLISP was brought up. Though it comes with a marvelous programming environment, its lack of speed and address space problems made it impractical to use. Once BLISS was chosen for SIGMA, all code was written in it. Perhaps a better scheme would have been to write an aim-high interface, in the IFCP manner, to exist between forks written in BLISS and others written in INTERLISP. Then aim-low modules, perhaps written in INTERLISP, could have been put together very quickly. Using this strategy implies a strong commitment to this design philosophy.

We all guess wrong (Axiom 8) but perhaps the main reason that this theory is not in general use is that people don't like to reprogram modules they have coded before. But a large software project has many people on it, so the solution is obvious. Let the aim-low implementation drive the design; when aiming higher, let someone else implement it. The daemon and FM rewrites were perfect illustrations of this point.

As with top-down programming, egoless programming, or chief programmer teams, no magic is offered. This theory is not a panacea for all software problems. Lousy designs and poor programmers will still defeat any methodology. What we do offer is a guess as to how to make a long-term project less painful and more rewarding.

REFERENCES


This research was performed for the Advanced Research Projects Agency under Contract No. DAHC 15 72 C 0308, ARPA Order No. 2223. The views and conclusions expressed in this paper are not necessarily those of any person or organization except the author(s).