Integration Engineering Perspective

Don O'Neill

IBM Federal Systems Division

Integration engineering is the process of creating a coherent system from its component parts, both hardware and software, such that at each stage of integration the evolving system exhibits increased functional capability, while remaining under firm intellectual control. Integration is a major managerial, technical, and logistical task, requiring the combined efforts of people skilled in system engineering, hardware engineering, software engineering, and integration engineering. In large systems software may be the principal integrating element. This paper discusses integration engineering approaches including a software—first strategy, simulation approaches, and an incremental release strategy as well as a project example.

IDEA OF INTEGRATION ENGINEERING

The large-scale computer systems developed by the IBM Federal Systems Division (FSD) typically involve overall system management and integration responsibilities that require contributions from multidisciplined teams. In fulfilling its integration role, FSD produces both hardware and software products that serve as major components within these systems. These products include processors, displays, mass memories, data buses, and special electronics, as well as a variety of executive and application software, fault detection and location software, and program generation and development software.

Integration engineering is the process of creating a coherent system out of its component parts, both hardware and software, such that at each stage of integration the evolving system exhibits increased functional capability, while remaining under firm intellectual control.

For large-scale systems, integration is a major managerial, technical, and logistical task, requiring combined efforts of people skilled in system engineering, hardware engineering, software engineering, and integration engineering. To blend these efforts into an integrated product, the people and their activities must be integrated, as depicted in Figure 1. FSD integration engineering practice accomplishes this through an incremental development approach that produces early operating increments of the final product. Incremental development applies performance pressure on development organizations and support tools by requiring early product completion from the multidisciplined team. Furthermore, the approach provides early visibility to the product under development.

Many large-scale systems are heavily interface driven. That is, they respond to stimuli from users at terminals and consoles, as well as from hardware sensors and actuators within the system. Dozens, or even hundreds, of interfaces may be present. In such systems, software is often the principal integrating element. It is the software that permits various hardware devices to operate as a coherent system, by synchronizing activities, managing and controlling interfaces, passing and processing data, and keeping track of system status. Thus, the incremental buildup of operational capability must concentrate on interfaces among hardware, software, and users. Since the software for integration must be ready when interfacing hardware becomes available, software development must precede hardware availability. In this software-first approach, hardware interfaces must be systematically simulated.

HISTORICAL DILEMMA

It is not enough to begin software development earlier to be ready for hardware. Systems engineering definition should be complete, though it often is not. Software is bracketed between two engineering activities: systems engineering definition and hardware availability. Both these activities may represent risks in meeting scheduled milestones.

System definition completion may be uncertain because in defining a solution to a new problem there is

Address correspondence to Mr. Don O'Neill, IBM Federal Systems Division, 18100 Frederick Pike, Gaithersburg, Maryland 20879.

The Journal of Systems and Software 3, 77–83 (1983)

0164-1212/83/010077-07$3.00

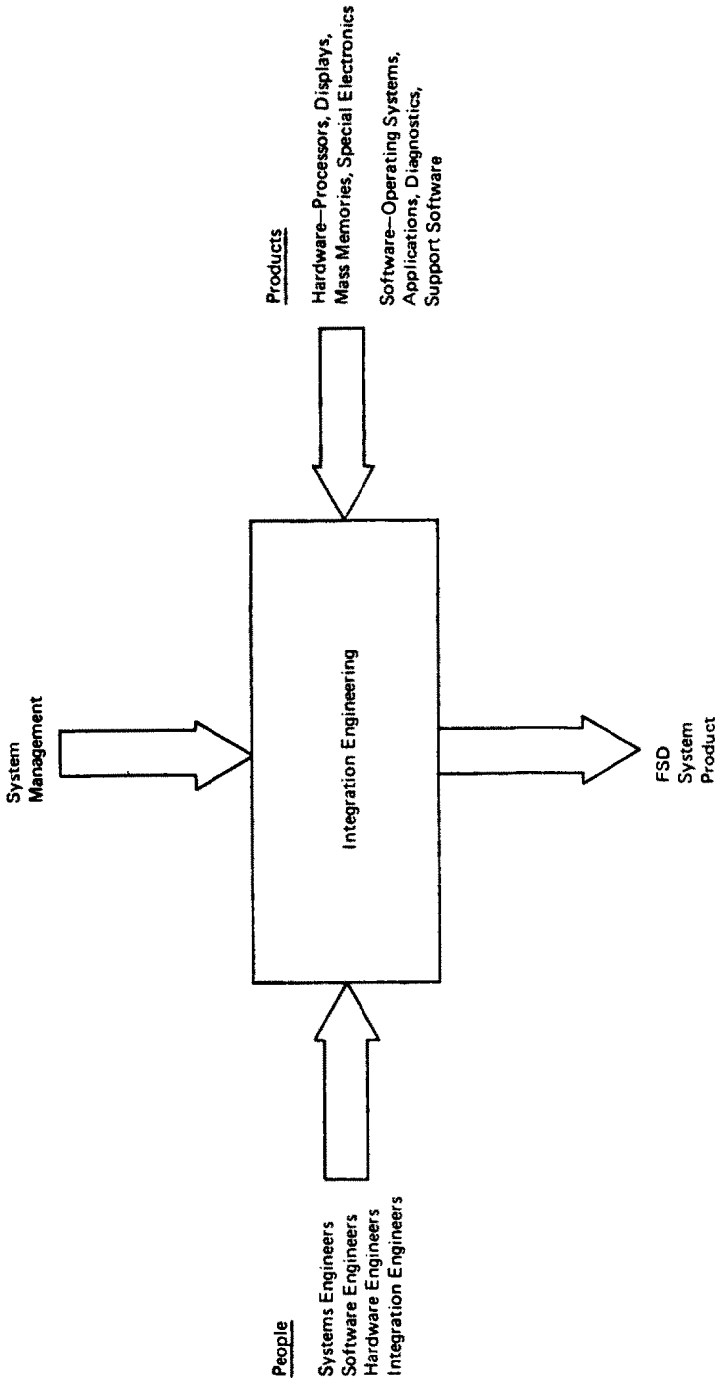

Figure 1. Blending people and products in integration engineering.

no precise precedent from which to plan. Definition is a creative task and so may not easily be scheduled into an orderly process. Furthermore, it is often difficult to define completion criteria for the system definition activity, and so software design may be forced to begin with incomplete definition.

Hardware availability has traditionally determined software schedules. When hardware is scheduled to be in place, software is expected to operate to support integration and test. Furthermore, completion of software testing is fully dependent on the availability of a processor and external hardware. External hardware is defined as any hardware that interfaces as an input and/or output device to the processor. Programmed instructions are developed for the processor. External hardware may include sensor equipment, operator display stations, and other processors. Figure 2 shows a typical configuration in which the processor interfaces with external hardware. Software development may not be scheduled with its natural period of performance, but instead may be paced by hardware schedules. If the hardware milestone is missed, the software effort may continue on a best-efforts basis without reduced manpower, even though it is not effective use of time and money. The successful software development requires that both complete system definition and hardware availability be satisfied on schedule.

SOFTWARE-FIRST STRATEGY

Hardware/software integration risk can be greatly reduced by recoupling software development from external hardware dependencies through hardware simulation. The software-first strategy reduces development risk associated with software and improves software manageability by eliminating its principal external dependency.

As noted, software is the principal integrator of component parts in large-scale systems. In particular, software must manage and control the many interfaces among hardware, software, and users. This role places software in a special position during system integration. In the absence of a counterstrategy, software was ordinarily the last system component to complete integration testing. Software testing depended on availability of target processors and external hardware. The software-first strategy greatly reduces this dependency by software simulation of hardware interfaces, so that software development can be performed in the absence of that hardware.

Of course, creation of software to simulate hardware can be a significant task in itself, one that comsumes project resources otherwise devoted to the primary software system. Yet the benefits of developing simulation software outweigh the costs. The principal benefit of "software first through simulation" is to decouple external hardware delivery from its dependence on the software development schedule. Additional benefits during testing include improved repeatability of test procedures in the software–to–software test environment. Tested software can be integrated with hardware with high confidence of correct operation. Since software capabilities typically include hardware fault detection, there is further reason to expect that hardware/software integration will be orderly. Furthermore, the process of developing software for hardware simulation is likely to provide a better understanding of hardware interfaces. Hardware engineers have a vital interest in ensuring that this understanding is complete and that all hardware specification updates are reflected in the

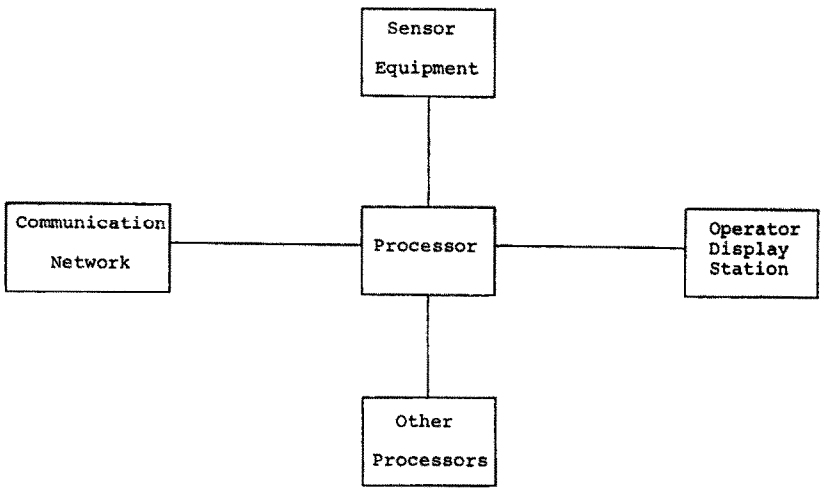

Figure 2. External hardware interfaces.

simulation software. For this reason, simulation software specifications should be configuration controlled as companion documents to hardware specifications. This makes the status of hardware development more readily visible to software management.

SIMULATION

External hardware may be programmed or nonprogrammed equipment capable of either input or input/output implying feedback. The author has had experience with three simulation approaches:

1. External interface via computational driver with no alteration of operational software under test;
2. External input via sequential device with some alteration to operational software under test;
3. Internally simulated interface with alteration to operational software under test.

Approach 1 is used where it is necessary to provide a high fidelity representation of the missing hardware by faithfully reproducing the input data content and arrival characteristics and providing a feedback loop to the driver function. This technique is required for complex hardware such as sensor equipment providing signal processing inputs. The benefit of this is that it enables the operational software under test to be used intact without alteration. The absence of the hardware is fully hidden from the software system under test which is execution equivalent to the operational software to be deployed.

Approaches 2 and 3 may be employed to derive the benefits of the software-first strategy in a cost effective manner. Approach 2 may be effectively applied in simulating an expensive and complex hardware element whose interface is limited to a sequential input stream by using an input data tape. Approach 3 may be applied in simulating a simple hardware element with both input and output interface and feedback by simulating the hardware in the target machine. Each approach has the disadvantage of altering the operational software performing the equipment control and the access mechanism within the target machine. Since any alteration detracts from the integrity of the test program, the use of subsequent regression testing is required when the hardware arrives. Approach 3 has the added disadvantage that simulation processing and feedback loop closure are performed on the target machine with distortion to computer loading measurements. These approaches provide functionally equivalent operation with respect to the operational software and lack execution equivalence in the hardware interface software. Furthermore, simulating the control functions of an operator display station using Approach 3 may be superior to the hardware itself because all manual input sequences can be exhaustively exercised in a repeatable way not possible with the hardware itself. Approaches 2 and 3 are very satisfactory software integration engineering solutions when certain types of hardware are missing.

INCREMENTAL RELEASE STRATEGY

The software-first strategy can be effectively combined with an incremental release strategy, so that early software integration is carried out through incremental releases decoupled from hardware development. This provides a management mechanism to appraise software development processes and procedures through early partial product delivery.

In performing software design for incremental releases, modern design practices require establishing hierarchical structures and performing modular decomposition. In this decomposition process, the functional content of each release can be systematically partitioned along well-specified interfaces. This helps reduce the complexity of development and integration by allowing developers to deal with a series of relatively small additions to an evolving software hierarchy, rather than forcing them to attempt integration of all the software at once. The result is more orderly integration and testing. Since errors tend to be localized to the last increment added, their detection and correction is simplified. More thorough testing of each release is the result.

An early release should include executive software, hardware fault detection software, and data recording software. Executive software is generally required in the initial release, since it represents the top of the system under development. Early delivery of software for hardware fault detection is useful in resolving hardware/software misunderstandings. An effective data recording mechanism implemented with a minimum of distortion to the software system provides a uniform data extraction interface to integration and test activities from the outset. Subsequent releases should focus on interfaces, then on gradual buildup of functional capabilities.

By implementing important functions in the early releases, design-to-cost options can be retained. This can be accomplished by rebudgeting the remaining functions into the remaining resources, following each release. In the event cost or schedule pressure develops, important functions will have been developed and any function reduction or elimination would affect less critical capabilities.

The incremental-release strategy provides an opportunity for improved management of changes. The intent is that each increment be developed against a firm set of requirements and specifications. Once a change

has been evaluated and approved, it can be assigned to an increment so as to minimize its impact on orderly system development. It may be possible, or even necessary, to incorporate certain changes in the increment currently under development, but most changes should be scheduled for future increments. Thus, nearly all changes to be incorporated in an increment will be known at the outset, so that designers and programmers can operate in a stable environment.

The incremental release strategy provides an orderly method for prototyping crucial elements during software implementation. Algorithms involving advanced technology for which in-systems prototyping is required can be assigned to early releases. Results can be evaluated, adjustments made, and a revised implementation included in a subsequent release. Although the incremental release strategy does support prototype testing, it is not intended to provide a cost effective test bed for algorithm experimentation.

Some important principles are necessary in applying the incremental release strategy. Functional contents of release increments must be natural, in terms of the problem, design, implementation, and test. This means functions to be delivered in subsequent releases should not leave an awkward gap. For example, an integral function should not be split between two release increments. A function should be delivered in its entirety in a single release. If a function is allocated two releases, there is the tendency to solve problems for the initial release, but to push the ultimate resolution forward to a subsequent release. This may be conscious or inadvertent, but in either case it may result in breaking previously delivered and tested code and in this way erode benefits of the incremental release strategy. When conforming to top-down structured programming, stubs will be used. It is not desirable to formally test stubs, except to be certain that when entered, exit is assured.

INTERFACE DEVELOPMENT

As noted, a complex system may be heavily interface driven. Hardware/ hardware interfaces require physical definition of electrical and signal characteristics; software/hardware and software/software interfaces require logical definition of data formats and information content. Both physical and logical interfaces must be documented and configuration controlled.

The software-first strategy is a natural approach to management and control of complex hardware/software interfaces, since it decouples software from dependency on hardware through interface simulation. Furthermore, the operational software will be developed incrementally with interface testing to be performed in the early increments. This strategy places additional early performance pressure on system engineers, software engineers, and hardware engineers to specify and design these interfaces so that simulation and operational software development can proceed.

SOME PROJECT APPLICATIONS

FSD has had extensive experience in the integration engineering approach to large systems development. One need only think of the Apollo, Space Shuttle, TRIDENT Submarine, and Light Airborne Multipurpose System (LAMPS) programs. For the TRIDENT Command and Control System, a series of incremental releases was used to integrate over a million lines of operational software. An early increment was planned to exercise and verify hardware/software interfaces. Unavailable complex hardware under concurrent development was simulated to permit the software to be developed first (see TRIDENT Integration Engineering below).

LAMPS development also featured an incremental buildup of capability, with primary focus on interfaces and end–to–end message flows. With user interfaces operating early, the customer could readily review development status, and assess the product design itself. These projects provided first-hand experience with incremental development, and the strategy is now well refined.

TRIDENT INTEGRATION ENGINEERING

The TRIDENT Submarine is under development to serve as a component of U.S. strategic forces. Its Command and Control System (CCS) is an integrated collection of complex electronic equipment supporting the submarine commanding officer and his staff in performing command, control, communication, defense, and ship functions.

The nerve center for the TRIDENT CCS is the Data Processing System (DPS), which includes four central processing units configured as a pair of multiprocessors. The DPS software, which exerts control over major system interfaces, is the primary disciplining influence upon overall system integration. The DPS software development is a significant responsibility, representing the largest tactical development ever undertaken by the Navy—approximately one million source lines of shipboard operational code. This software provides computational support for various sensor, equipment control, and command subsystems, each of which is the development responsibility of a particular Navy organization. These separate developments have been integrated into the DPS by FSD under contract to Electric Boat Division of General Dynamics.

The organizational complexity of the TRIDENT system development community required that each

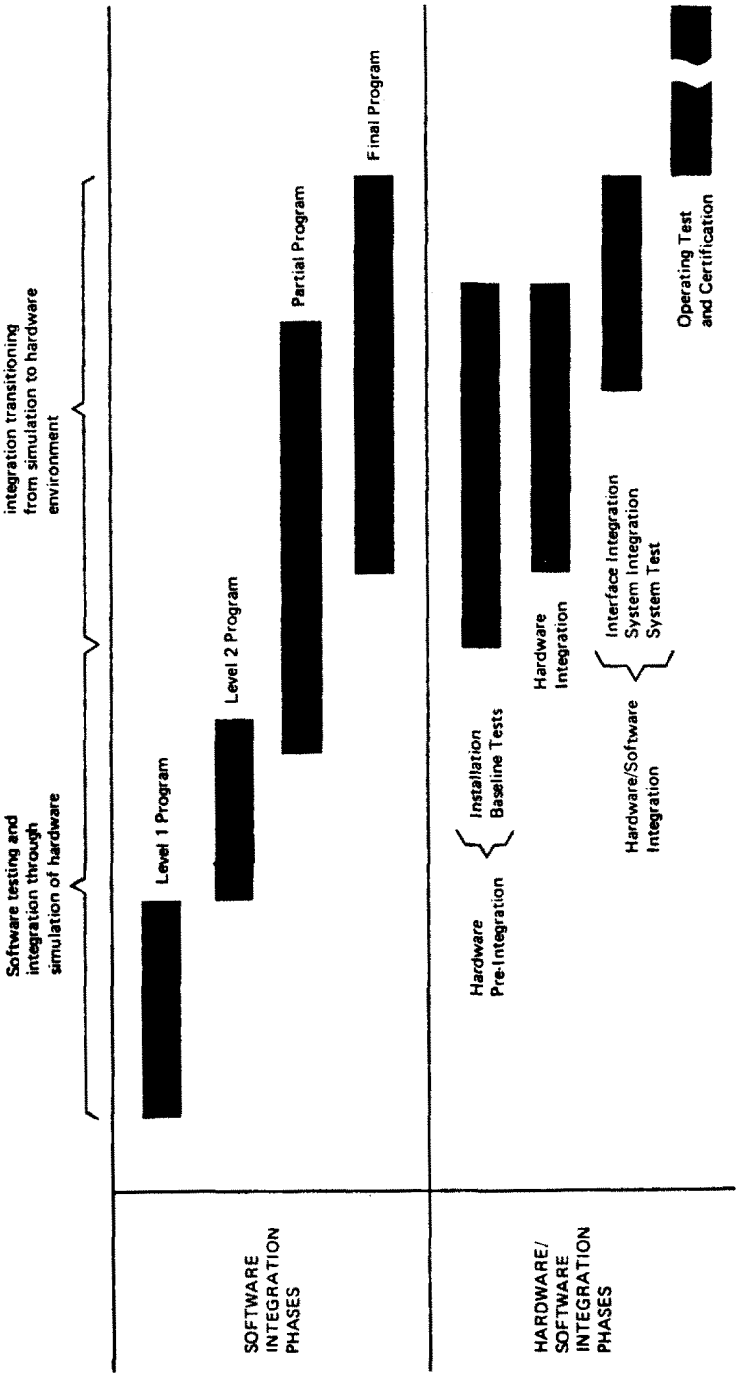

Figure 3. TRIDENT command and control system integration approach.

subsystem be developed as an individual entity of both hardware and software. To achieve ship schedule objectives, it was necessary to integrate CCS software concurrently with its development, and to drive the separate subsystem efforts from the top down. Thus, the hardware, software, and system development and integration schedules all ran concurrently. The separate development efforts, coupled with overlapped schedules for development and integration, demanded a close-knit integration effort if the schedule was to be met.

In order to meet the ship schedule an additional step was necessary—hardware and software development efforts were decoupled to allow fully parallel implementation. This was accomplished through extensive software simulation of hardware, in order to permit testing of subsystem software prior to hardware availability. Individual subsystem simulations were subsequently combined to provide further benefit as a software integration tool before the full complement of CCS hardware was assembled for certification.

In the incremental approach, programming begins with overall system architecture and control logic, proceeds next to interface definitions, and finally moves to functional and computational segments of code. Although functionally incomplete, software developed according to this technique is always executable; thus, integration and testing can be carried out continually during the development process.

The complexity of the TRIDENT integration problem demanded early management and technical focus, with suitable published plans and procedures in order to ensure success of the effort. The software integration plan called for sequential delivery of four separate increments, each building on and extending previous increments. The four deliveries were termed Level 1 Program, Level 2 Program, Partial Program, and Final Program. As shown in Figure 3, the Level 1 Program increment was scheduled for early delivery. Level 1 software was intended to validate DPS subsystem resource utilization and to exercise interfaces between subsystems and the executive control program and other system software, as well as to confirm the procedural discipline of the integration process itself. Each subsystem simulated its eventual architecture in coded stub form. Simulated resource allocations within each stub included CPU time, core storage, channel usage, and disk storage and access requirements. Each stub was in fact a DPS resource specification that invoked resource simulation support software during execution. In this manner, anticipated software architecture for each subsystem was exercised dynamically, with results of resource utilization recorded for later analysis.

The Level 2 Program increment was intended to be a thorough test of subsystem interface definitions, including interfaces to data acquisition, formatting, quantizing, and error-processing software. An interface design specification formed the basis for subsystem software development, as well as for test planning and test procedure generation. Simulation software provided system inputs in the form of high-fidelity signal representations in place of unavailable subsystem hardware. Successful completion of Level 2 integration marked a significant milestone, since it demonstrated DPS support for all subsystem interfaces. This is a crucial step in the integration process; verification of system control and interface design provides a sound framework for subsequent addition of functional software.

The Partial Program increment added the minimum set of critical functions required by the CCS. This software resembled certain degraded mode capabilities that the total system would eventually possess. Thus, a coherent and operable software system was scheduled for early delivery and received extensive testing throughout the remainder of the development process. In addition, this delivery formed the foundation for the Final Program increment, so that the benefits of continued testing within an integrated system were retained.

The Final Program increment included all remaining functions to be developed; its integration was completed by using simulation software. Following the Final Program phase, simulation software was gradually replaced by subsystem hardware during a period of hardware/software integration.

In addition to providing the functional processing capability required, DPS software incorporates a measure of fault tolerance to hardware errors, including both transient faults and solid failures. Defensive programming techniques protect the user from effects of transient faults through both software retries of hardware operations and warm starts. Solid failures require transition to degraded mode operation within seconds, possibly with loss of functional capability, but with continued operation assured. In effect, this fault-tolerance software has enhanced system reliability beyond the reliability of individual hardware components in the system.

CONCLUSION

As the software process is better understood, and the role of software within large systems is more clearly defined, innovations will emerge to improve software performance. The software-first strategy and the associated incremental release strategy represent such innovations and are important elements in the foundation of software integration engineering.