EDP ANALYZER
© 1979 by Canning Publications, Inc.

DECEMBER, 1979
VOL. 17, NO. 12

PROGRESS TOWARD SYSTEM INTEGRITY

In the computer field context, the term 'system integrity' can be defined as the behavior of a hardware/software system that "does the right things and *only* the right things; moreover, the system does those things right and does them when they are needed to be done." System integrity is a goal to aim at, and one that can never be fully realized. But it is an important goal, even in a business data processing environment. Imperfections in today's computerized systems all too often lead to frustrations, annoyances, or even harm. Some pioneering work in secure operating systems may be pointing the way for achieving a higher degree of system integrity in business applications.

In 1973, SRI International (formerly Stanford Research Institute), of Menlo Park, California, received a contract from the U.S. Department of Defense that had two main goals. One goal was to design a highly secure operating system. The second goal was to be able to prove that this system was secure. In order to permit formal (mathematical) verification of system properties, such as its security, SRI developed a rigorous methodology for system development.

The problem that SRI was tackling, as described by Neumann (Reference 3a), was: how should one design, implement, debug, operate, modify, and maintain a large, complex computer system that includes both the hardware and the operating system? Further—and this was a critical part of the methodology—they wanted to be able to formally verify that the system actually would do what its specifications say that it should do, for such things as performance, security, reliable operation, and recovery from faults.

As we have indicated in previous reports (for instance, May 1970, July 1977, February and March 1978, and January and February 1979), developing systems that do what they are supposed to, and only what they are supposed to, poses very challenging problems. It is difficult to determine requirements, then to develop specifications for the new system that meet those requirements, and then to build the new system to meet the specifications. Once the system is built, the problem does not end; it must be maintained and modified without destroying its desired characteristics.

The SRI work has drawn on the results of a number of other projects attacking this prob-

ISSN 0012-7523. Multiple copy prices listed on last page. Photocopying this report for personal use is permitted, providing payment of $2.00 fee *per copy of report* is made to Copyright Clearance Center, P.O. Box 765, Schnectady, N.Y. 12301; please include identifying fee code 0012-7523/79/120001-13$02.00.

lem area, including work done at Ford Aerospace, the MITRE Corporation, and the University of California at Los Angeles. In addition, similar work has been going on in other countries. Most of this work is not classified from a military security standpoint. The SRI people have had a lot of information interchange with these other projects, particularly with the MITRE project. Even though each project has followed a different approach to the problem, the SRI people have benefitted from this interchange.

The major results of the project to date are as follows, as described in Reference 1:

Hierarchical development methodology (HDM). This is the methodology for developing systems that supports formal verification of system operation, in accordance with the specifications. It, in turn, is made up of several components.

Concepts. The methodology uses *hierarchical decomposition* for attacking a complex problem. It also uses *abstraction*—isolating a few properties of an object that are appropriate for explaining or understanding that object, at the particular level of the hierarchy under consideration. It uses *modularity* which sub-divides the system into easily replacable parts that have well-defined external interfaces. It uses *formal specification* via a somewhat-mathematical specification language. Unlike natural language, this specification language is unambiguous—each statement has one and only one meaning. The method employs *design* verification, to prove that specifications are consistent with formal requirements. It also uses *program verification* for determining the consistency between a program and its specifications.

Procedures. HDM divides the development cycle into eight stages, starting with conceptualization and an informal statement of the problem, and ending with the production of verified code. It is being extended to cover subsequent incremental changes for improvements and maintenance. Note that conventional program testing and debugging is greatly reduced; the formal verification process performs the same function much more effectively.

Languages. HDM uses three specification languages. SPECIAL is used for specifying and representing modules. The operations that the module must perform are specified independently of how they will be performed in the implementation. Also, HSL is a language for describing levels and hierarchies of levels. Finally, ILPL is an intermediate level programming language, used for describing abstract programs. Alternatively, a modern programming language such as Ada, Modula, or Euclid, may be used directly.

Tools. The people at SRI have developed prototype tools for several of the development stages. These tools include a module checker, a representation checker, an interface checker, and a hierarchy checker. Several other tools are either in development or have been proposed.

HDM has been used on a number of projects dealing with complex systems. These include three projects on secure operating systems, one on a highly reliable flight control system, and one on a family of real-time operating systems.

As an example of the use of HDM, we will briefly describe one of the secure operating system projects.

Provably secure operating system (PSOS). At a session of the IEEE Computer Society Compcon Spring 79 conference, E. L. Burke of the MITRE Corporation, gave his views on the current status of secure operating systems (Reference 2). The first generation of such systems has been characterized by the adoption of sound engineering principles to the development of software, he said. The approach includes top-down design, the use of design and implementation tools, and the use of a 'somewhat simple' formal model of the security problem. But first generation technology had to limit the scope of the problems that were tackled, he added. For instance, non-security aspects—such as denial of service—were skipped. Also, the formal model did not include the hardware, allowing the possibility for hardware maintenance people to gain access to the security system. And the tools used were relatively independent and stand-alone, rather than being an integrated set of tools for design and implementation.

The second generation of secure operating systems is just beginning to appear, said Burke.

Some of the shortcomings of the first generation are being overcome. And one example of a second generation secure operating system is PSOS.

As discussed by DeLashmutt (Reference 2) and Feiertag and Neumann (Reference 3b), PSOS has been designed to be a secure operating system that is independent of specific hardware/software boundaries and of specific hardware. However, the hardware must be able to support the concept of *capabilities*, to be discussed shortly. The capability concept can be used equally well as the protection mechanism for user programs, application sub-systems, and system software. No special protection is needed for system software.

Further, PSOS is designed to provide multi-user, multi-security-level service. As an example of one of the first generation shortcomings that it has overcome, it protects against one user getting into the domain of another. And it protects against a user being able to see residues in memory left by previous users of the memory resource. PSOS has many other similar features.

Feiertag and Neumann report that the design of PSOS has been formally specified, using SPECIAL. The specifications define PSOS as a collection of about 20 modules, organized in a hierarchical structure. The design has been completed to the point where the features of PSOS can be compared with those of other secure operating systems.

PSOS has not yet been implemented. One reason is that no single, commonly-available computer has all of the features needed for implementing PSOS efficiently, although all of the required features do exist in some hardware.

As an example of a hardware support feature needed by PSOS, consider the concept of a *capability*, as used in PSOS. In order to gain access to an object (such as data or another program) that is controlled under PSOS, a user program must present an appropriate capability to the module that is responsible for that object. A capability is a token (a string of bits) that includes (1) a tag, (2) a unique identifier, and (3) a set of access rights, telling what operations the user program may perform on the object. The tag is critical; it distinguishes a capability item from, say, a data item. The hardware must make the tag field inaccessible to programs, so that capabilities cannot be forged or altered by user programs.

The formal techniques used in the PSOS design both make implementation straight-forward and make the formal verification of correct operation possible. So the PSOS design should lead to much more secure and reliable operating systems than are now commercially available, say the authors.

In short, they see HDM as providing a way to build software that does everything its specifications say it should do, and nothing it should not do.

The problem of system integrity is pervasive in computer-based systems. It appears to us that projects such as this one at SRI may be charting a path that *many* system developers can eventually use. In order to build highly secure operating systems, they must find a way to develop software (to operate in a specific hardware environment) that does exactly what it is supposed to, and nothing else. In most business applications, the requirements will not be as severe. But the principles for achieving software integrity will still apply.

The need for system integrity

Definitions of the word 'integrity' in dictionaries usually include something like "the condition of behaving justly, properly, and honestly, according to standards of good behavior" and give as an illustration the phrase *a man of integrity*. In the computer field, the term seems to be quite widely used but with several different connotations. 'Data integrity' generally means the "the condition of being whole, complete, accurate, and timely," for instance. And the term has been applied also to both programs and systems.

Our use of the term 'system integrity' will mean what we indicated at the beginning of the report—namely, the behavior of a hardware/software system that does the right things and only the right things; further, it does these things right and does them when they are needed to be done. We should note, however, that not only is this term not widely used in this manner but also that there is some controversy about it. The controversy centers on whether this is a valid use of 'integrity.' But we feel that a system can exhibit integrity in much the same way that a person can, so that 'system integrity' is a valid concept.

Neumann, in Reference 3a, uses the term 'system defensiveness' to denote a part of what we include in 'system integrity'—namely, the non-existence of inappropriate behavior, as much as possible. He says that the components of defensiveness include security, reliability, availability, recoverability, and auditability.

System integrity is particularly important for systems with severe operating constraints. Some of these include air traffic control, the control of nuclear power generation, vote counting, and certain law enforcement systems, where errors of operation can have very serious consequences. Similarly, the security of classified military information is very important; when such information is stored in computer systems, the need for secure hardware/software is evident.

The business data processing environment has a need for system integrity, but usually not as extreme as the cases just cited. The need for data security is becoming recognized, and will become even more significant as countries pass privacy laws that control the transfer and disclosure of information about persons. And, of course, as organizations begin to store sensitive information such as company plans, trade secrets, etc. on computers, and control the transfer of their funds by computers, security will be essential.

But even in more mundane applications, such as billing, system integrity is needed. And the history of computerized billing systems certainly indicates that their integrity has not been as high as might be desired. Systems that continue to send erroneous bills (and adding more penalties each time), or dunning for zero balances, have undermined some of the public's respect for computers.

One might then ask: What is so difficult about getting systems with an adequate level of integrity?

Some of the problems

In our previous issues on this general subject, listed earlier in this report, we have given some idea of why it is so difficult to achieve system integrity. We will briefly review some of the causes.

The life cycle of a system has the following main components: (1) a new system begins by considering the mission of the activity that the system is supposed to serve—what really should the system do, and why; (2) then comes the requirements for the new system; ideally, the requirements define what the system must do, in support of the mission; (3) the specifications for the new system follow; these are (formal) statements of what the new system must do, based on the requirements; (4) next is the design of the new system, which should be in harmony with the specifications; (5) the implementation of the new system follows; it should be in harmony with the design; (6) next is the test of the new system, to assure the developers and users that it does what (the mission? the requirements? the specifications?) say it should; (7) then there is the maintenance of the new system, to remove detected errors; and finally there is (8) the enhancements and changes made to the new system, during its life.

Studies have shown (as we reported in the previous reports) that many of the problems with new systems can be traced to errors in the requirements statements. The errors include incorrect requirements statements, missing or incomplete statements, unclear or ambiguous statements, and inconsistent or incompatible statements.

So a part of the problem of obtaining system integrity has to do with the inadequacy of methods for determining and stating requirements. We discussed some improved methods for doing this in our January and February issues of this year.

Requirements errors often are not detected until the new system has been built and is being tested. It would be much easier and less costly to correct them had they been found by the specification or design stages.

Another part of the problem comes from moving from one stage to the next. In going from requirements to specifications, some additional errors may be injected. The same is true in going from specifications to design, and so on through the whole life cycle.

In short, almost regardless of how talented or experienced the development staff is, errors will creep in. These are errors both of omission and commission. Methods are needed to flush out the errors as soon as possible.

Also, the problem of shortcomings in system integrity can apply to just about all computerized systems, from quite small to large. One relatively small interactive system that we have studied, for instance, required about five man-months to design, construct, and debug. The user has been quite happy with the resulting system, which has been in daily use for over two years. But, in actuality, the system has had some integrity faults. There are a number of functions that the user requested which the system does not do. And in a few instances, the system does things that it is not supposed to do. Some of the shortcomings were due to an inadequate statement of requirements, according to the user. But the types just mentioned—system not doing what it should and doing things it should not—probably were caused by the use of an inadequate methodology.

The methodology used, in this case, consisted of the user supplying the implementer with a fair amount of informal documentation on requirements, followed by a greater amount of verbal communication. No *formal* requirements statements were developed, nor were specifications developed. Design and programming drew on the requirements documents and verbal communications.

This approach is not untypical today, we suspect. Some larger users are attempting to put more formality in the development process, as we have discussed in other reports. But many of the medium size and even larger size organizations are still using an approach similar to the one just described. And as the use of micro-computers spreads through smaller organizations, the same type of thing will happen—again and again.

At the other end of the size and complexity spectrum, Neumann (Reference 3a) discusses some types of system design and construction practices that have been found to cause security flaws in operating systems. Some examples are: a poor choice of security system boundaries, thus allowing users to get at security-critical functions; users allowed to use absolute input-output addresses; operating systems that assign two different names to the same object, or the same name to two different objects; and a lack of validation of critical conditions and operands, such as outside-of-bounds input values.

These and similar problems can be traced to deficiencies in the development methodologies used, says Neumann. Such deficiencies include the lack of formally stated requirements, formally stated specifications, formal proof of correspondence between specifications and requirements, and so on. So even though operating systems have usually been developed by very capable people, the same general types of shortcomings mentioned above for a small system are found to occur in many large operating systems.

Neumann compares two of today's operating systems, MULTICS and UNIX, for security features. MULTICS was designed (in the mid-1960s) to provide strong security features; UNIX was designed in the early 1970s for use in a benign environment and makes little pretense at being secure. (Neumann said he ignored the conventional commercial operating systems in his evaluation because they are, for the most part, intrinsically insecure.) He concludes that MULTICS is relatively secure, but points out some areas of possible weakness. UNIX, on the other hand, is not secure—apart from a basic set of read/write protect bits. This is not a criticism of UNIX, he says, because it was not designed to be secure. Rather, he simply wants to show how insecure an operating system can be in the absence of a real concern for security during its development.

Neumann made the point to us that security is only one aspect of what we have called system integrity. Some aspects of integrity will be imporant in every system; security may or may not be an important aspect in a particular case.

In general, it appears that *any* desired aspect of system integrity—performance, reliability, security, availability, etc.—must be actively sought by the developers. It will not happen as a matter of course or by assuming that "since we are using only top-notch people on this project, we won't have to worry about system integrity." System integrity is a generic problem that applies to all systems, from small to large, and from relatively simple to horribly complex.

Elements of system integrity

About two years ago, a committee of the American Federation of Information Processing Societies (AFIPS) organized a by-invitation-only workshop to explore the subject of system integrity. As the ultimate objective, AFIPS hoped to have a 'best practices' manual on system integrity developed to their specifications, along the lines of the AFIPS Security Manual. The workshop was asked to explore the question, consider whether such a manual was doable, indicate the type of coverage that might be expected, and suggest how an author search might be conducted.

While additional work was done after the workshop was held, for a number of reasons the project has been put into a 'hold' condition. Hopefully, the project will be re-activated; we think such a manual is both important and needed.

For discussing the elements of system integrity, we will draw upon the (unpublished) report of that workshop. We should mention at the outset that the workshop participants saw two primary audiences for the proposed manual: application system designers, and managers (both of data processing and of user departments). Also, included among the participants were several people from the business data processing field, so common DP applications, such as billing, were within the subject area that was discussed.

The workshop participants viewed system integrity as consisting of a number of factors or elements. A system should be *available*, ready to serve users when the users want to use it. The system should be *appropriate*, in that it does the right things, and *bounded*, in that it does only what it is supposed to. And the system should be *correct*—what it does, it does right. The system should be *predictable* by always doing things the same way, and should be *timely* by doing things at the right time and delivering results when needed. The system must be *maintainable* and not lose integrity in the process of being fixed or enhanced. And it should be *auditable*, so that auditors can verify the integrity of the system. The workshop pointed out that these factors seemed to cover such concepts as reliability, security, recovery, change control, and so on. Further, the factors are inter-related, so that trade-offs often will have to be made among them.

These factors perhaps indicate the 'ideal' in system integrity. But the workshop participants felt that levels of system integrity may have to be considered, from a practical viewpoint.

Levels of integrity

Intuition says that the concept of levels of system integrity is a valid one; system integrity is not an all-or-nothing matter. For example, an on-line word processing system serving, say, ten user stations does not need the same degree of availability, correctness, predictability, and timeliness as does a control system for a space probe.

So levels of system integrity are determined not only by which of the elements listed above must be emphasized but also by the degree of emphasis given each one. For a control system with high reliability requirements, not only should availability be emphasized, it probably should be considered a critical factor, and the cost of providing it probably should not be given the same weight.

The workshop participants felt that the levels of system integrity might be partially determined by the tightness of coupling between the system and its end users. For instance, some batch systems can be down for hours without end users being aware of the problem. But if an on-line sales order entry system goes down, terminal operators are aware of it immediately.

Another aspect that would seem to influence the levels of system integrity is important social values. Computers are being put into systems where faulty performance is intolerable. Examples are air traffic control and the control of nuclear power plants. But even within the business data processing environment, social values must be considered. As indicated earlier in this report, the designers of a good many of the computerized billing systems did not adequately consider social values—the frustrations, the annoyances, and even the actual harm that their systems have caused—when they designed the systems.

In fact, end user satisfaction or dissatisfaction with a computerized system is really a social value. In this sense, the designers of all computer-based systems should be concerned with social values, such as the threshhold where user procedures cease to be acceptable and become frustrating. System software often provides examples of disregard for social values—the frustration users get from supplying input to some operating systems is exceeded only by their frustration trying to decipher output error messages.

Higher levels of system integrity cost larger amounts of money to accomplish. So system developers must determine the appropriate level for a new system. And this does not appear to be easy to do, especially since humans are involved in *all* systems and the human element makes system integrity less predictable and controllable.

End user involvement. In order for system integrity to be maintained, there are some things that end users must perform. They must know system limitations and not try to force the system beyond its limits. (Hopefully, of course, system designers should check all user interactions for outside-of-bounds values and protect against any such.) But even more important, they must understand their role in the overall system and its relationship to system integrity.

During system development, it is essential that the developers obtain end user participation for determining what the system should do and what it should not do. For this, they need contact with the *real* users, not intermediaries. Obtaining this type and amount of end user support is not always easy.

In short, end users have an important role to play in achieving an appropriate level of system integrity, both during the design and construction of the system as well as during its operating life.

Life cycle considerations. Application systems change and evolve through time; they do not remain static. We have been told of studies that report about 80% of development staff costs, on the average, are spent on maintenance and enhancements of application systems. The original development represents only 20% of the total. Further, other studies have pointed out that the original structure of a system tends to be destroyed by changes and enhancements.

What this means is that system integrity cannot be considered just during system development and then put aside. The desired level of system integrity must be maintained during the whole life cycle of that system.

It appears to us that the concept of levels of system integrity is valid. Designers must select the appropriate level for any given system. The appropriate level is a function of the tightness of coupling between the system and its end users, as well as of the social values held by the users. The effects of the behavior of both users and maintainers on system integrity must be considered. And in determining the appropriate level of integrity, the cost of both providing and maintaining that level over the life cycle of the system must be determined. These are not easy things to accomplish.

A project, such as the one contemplated by the AFIPS committee, might be able to locate practical methods for accomplishing these things. It would be very useful, we believe, to have such methods collected and published.

But computer-based systems are being built, and in rapidly increasing numbers. Their designers typically would like to meet the (probably poorly stated) integrity goals. In the face of all of the above factors, how might system developers achieve a desired level of system integrity? A part of the answer is: use formal methods.

Use of formalism

Formal methods become more and more important as the severity of constraints on the system increases. At the least, formal methods require written documentation, formal review techniques, and standard approaches. The standard approaches, in turn, involve standard project phases, standard documentation for each phase, and standard procedures to make sure that all detected errors have been properly corrected.

As the discussion of HDM has indicated, formal methods move into the realm of mathematics as the severity of the constraints passes some threshhold. That is, the design of a highly secure operating system (probably) cannot be accomplished without the precision of mathematical tools.

Many system designers in the business world are not too proficient in mathematical methods, we suspect. So mathematical formalism, of the type used in HDM, say, will have to be 'translated' into a form more appropriate for such designers. That has yet to be done. Because the need exists and the rewards for accomplishment will be substantial, we believe that it will be done. In the meantime, those system developers who *can* handle the mathematical procedures of HDM (or other such methodologies) may well decide to begin using them.

Before outlining some of the components of formalism, it would be well to mention a couple of problems related to it. One problem is the apparent delay formal methods seem to cause—the 'huge' increase of time spent in the requirements, specifications, and design stages of a project. It is not unusual for some people to finally say, "Let's cut out all this foolishness and begin writing some code." (Neumann points out that implementation and maintenance time savings more than offset this delay.)

Another problem area is that formalism can become bureaucratic, especially if the people do not understand the basic concepts but only know the procedures to be used. Then the concern is more with form than with substance.

It seems to us that both of these problems, and other similar ones that might arise with the use of formal methods, can be adequately dealt with by an interested and concerned management. But this means that management must understand what the formal methods are attempting to do, and why. This is a problem area in itself.

With this background, let us now look at some of the components of formal methods, as identified in the workshop.

Needs assessment. Formal methods already exist for determining what the new system must do. For instance, we discussed SADT and IA in our January issue of this year. Both of these are graphical languages, and have the advantage of communicating well with end users without the use of jargon or mathematics. Having determined requirements and documented them with such a language, it *might* then be desirable to state the requirements in a mathematical-type language, for later proving correspondence between requirements and specifications.

Another need exists at this level—that is, a method is needed by which the appropriate level of system integrity can be determined. To our knowledge, this question has not been addressed by any of the methodologies we have encountered. The problem is either ignored (in most cases), or else maximum integrity is sought.

Formal specifications. We gather, from our discussions in the field, that many of the shortcomings that have occurred in computer-based systems could have been avoided through the use of formal specifications. These specifications state, from the designer's point of view, *what* the new system must perform. They represent the designer's understanding of the user's requirements.

HDM provides a specification language, SPECIAL. By using this language for stating both the requirements and the specifications, it is possible (claim the developers) to *prove* the correspondence between the requirements and the specifications. It seems to us that this facility provides a big step forward for system integrity.

Modes of failure analysis. In addition to telling what the system *should* do, the specifications should also fully describe what it *should not* do. In other words, the system analysts and system designers should make a thorough analysis of the ways that the system might (a) not do what it is supposed to and (b) do what it is not supposed to. The specifications should then specify actions to guard against these possibilities.

Formal design documentation. Correspondence between specifications and design must be established, in much the same manner as between requirements and specifications. Formal design documentation will help to accomplish this.

Hardware/software selection. In order to meet the appropriate level of system integrity, the hardware, operating system, and purchased software must be considered. A manual, such as the one contemplated by AFIPS, might provide a checklist of things to look for, plus some 'good practices' that might be followed when selecting hardware and software.

In many (most?) instances, of course, application system developers will have little or no say about the hardware or operating system that is to be used. But, at least, if the deficiencies of the hardware/software are known, this would reduce the chance of unrealistic expectations about the overall system integrity. It would also indicate what hardware difficulties the software should attempt to overcome.

Test plans and quality assurance. A method for verifying that the completed system agrees with its specifications is essential. The most widely used method for this, by far, is testing—that is, provide a variety of types of input and see that the proper outputs are produced.

During the design and construction stages, this testing can be simulated by means of design reviews and code inspections. The developers can 'walk through' a variety of input types, for a small group of reviewers, and indicate what outputs will be produced. This is a slow process, and can be used for only the major input types, we gather.

The formal verification methods used in HDM, however, apparently bypass much of this conventional testing. By logically demonstrating that a program does exactly what it is supposed to, and nothing else, the need for testing that program is greatly reduced. Formal verification methods are very new, require an ability with mathematics, and are rather difficult to apply. Where the need exists for a high level of system integrity, however, the use of formal verification may be essential.

Formal training. Since human behavior can have such an effect on system integrity, the users of the system should be trained in what to do and what *not* to do. They should understand what the system limits are and should be encouraged not to try to force the system to operate beyond those limits.

The training program is needed not only at the outset, when the new system is being installed, but also must be repeated as new people join the organization.

Formal turnover. As the level of needed system integrity increases, so does the need for a formal procedure to move the system from development to production status. There would be a number of things that the production people would want to check before accepting the system. Further, this formal turnover should insure that no changes are made to the system that have not gone through formal change control.

Failure analysis. Achieving and maintaining a high level of system integrity requires that all system failures be recorded, analyzed, and corrected. Formal procedures are required for this, as well as for making sure that the corrections have been made, made correctly, and installed.

Auditing. Financial audit procedures tend to be formal in nature, in that well-defined procedures are used. However, audits for security, privacy, and integrity features are not yet as well defined; a definition of their audit procedures is needed.

It also should be mentioned that the integrity features should be auditable, to make sure that they have not been compromised. Ideally, this means that there should be some way of verifying that the object programs being used are faithful translations of the source programs and that the source programs correspond with their formally verified designs, and so on back to requirements.

Change control. There is probably no aspect of system integrity that is more important than change control. As indicated earlier, some 80% of staff time goes into maintaining and enhancing existing systems, over the life of these systems, and only 20% goes into the original development. When changes are made, errors can creep in (or can be implanted), which compromise system integrity.

So formal change control procedures should be used, for reviewing, approving, making, and verifying all system changes.

The change control procedures should apply to *hardware* (CPUs, memory, peripherals, communication equipment, etc.), *software* (operating system, utilities, application programs, and so on), *programming languages used,* as well as the *procedures* used for developing, maintaining, and operating the systems.

It should be recognized that this need for change may be caused by changes in the operating environment. These include new applications, growth or decline in business volume, mergers or acquisitions, restructuring of an organization, change in company goals, markets, or operating policies, and so on.

Here, then, are at least some of the types of formal methods that could be used for achieving a desired level of system integrity. In a sense, one can feel discouraged by all that is needed in order to achieve system integrity. However, we suspect that the various components will be packaged and sold commercially not too many years hence, with a good number of automated support tools included.

Even from our brief summary, it should be apparent that not all of the needed formal methods yet exist. But a good start has been made toward developing these formal methods. To illustrate this, let us now look more closely at HDM. It may be the forerunner of the 'high integrity' development methodologies we anticipate.

HDM

As mentioned above, HDM was conceived for developing large, complex software systems where a very high degree of system integrity is required. An example of such a software system is a multi-user, multi-security-level operating system. But the methodology can also be applied for the development of smaller and/or less stringent systems.

HDM is a sophisticated methodology that is based on scientific principles—principles which include the concept of data abstraction and the mathematical basis of programming.

Our discussion here draws upon a report by Levitt, Neumann, and Robinson (Reference 4) on the use of HDM for developing software. The language used and the example that is described in that report are appropriate for the intended audience—namely, designers and implementers of large, complex software systems such as operating systems. People who are involved in the development of business application software may find the report a bit difficult to translate into familiar terms.

Following are some of the principles upon which HDM is based.

Hierarchy of levels. HDM uses hierarchy for handling complexity. It calls for the decomposition of the design of a new system into a series of levels. The top level is the user interface. The bottom level (generally) may be the hardware upon which the system will run or the programming language in which it will be written.

Abstract machines. The methodology views the new system as made up of one or more 'abstract machines' at each level of the hierarchy. The term 'abstract' is used to indicate that only those details that are appropriate for a particular level will be considered; all others will be hidden at that level. The term 'machine' implies that, in theory, a mechanism *could* be conceived that performs at that level. For example, one might conceive of a hardware/software complex that had to be told only to "run payroll," and it would be able to determine everything that had to be done. We are *not* referring to the case where the whole payroll system has been programmed in the conventional manner; we are postulating the case where the hardware/software figures out what has to be done. The lower level abstract machines specify the 'what has to be done.'

At the top of the hierarchy, the abstract machine deals with what is supplied by the user and, in turn, what must be delivered to the user. At the bottom level, the abstract machines deal with how those user services will be provided by the computer.

Modules. Each abstract machine consists of one or more modules, where a module is a programming unit that is independently implemented. That is, the internal details of a module are hidden from everything outside of the module. A module can call on a lower level module for service.

Stages. HDM consists of eight stages, divided into three main categories. The first three stages make up *design*, the next two make up *representation*, and the final three are *implementation.* Generally, the stages are performed in order, in the sense that all design decisions are made before moving on to the representation and implementation decisions. However, backtracking can (and probably will) occur.

The authors point out that HDM does *not* require top-down development. For instance, when considering a particular abstract machine, the designer might well also be considering the lower level abstract machines that will be needed to implement that machine.

Also, it is quite possible, they say, to do top-down design and bottom-up implementation.

Before discussing the HDM stages in more detail, several comments are in order. As mentioned earlier in this report, today's 'conventional' development methodologies typically use stages that are named: problem definition (or determining requirements), system specifications, system design, program design, coding, test, installation, and maintenance. The HDM stages are similar, but they do differ from these in important ways, as we will discuss. Testing is still used, but the need for it is reduced, and HDM is being extended into the maintenance area. Secondly, HDM uses the SPECIAL language for the design and representation stages, and ILPL (intermediate level programming language), or another programming language, for the implementation stages. Thirdly, correctness verification is distributed across stages 1, 4, and 7 of HDM, and is not just limited to a single stage of the project. And finally, the problem definition step (requirements determination) is not covered by the authors, other than the re-statement of the requirements in the formal language SPECIAL.

The design stages. The first three stages of HDM are concerned with design.

Conceptualization. In this stage, the intent of the new system is described in natural language. The services of the top-level module (the user interface) and the components of the lowest level module(s) are indicated. This conceptualization indicates that the design of the new system is beginning to take shape in the designer's mind. However, later stages may show the need to re-think this conceptualization.

Decomposition of the top and bottom levels into modules. It is possible that the top-level abstract machine, and/or the bottom level one(s), may only require one module each. In complex systems, multiple modules are more likely. As indicated earlier, each module is defined to be a 'stand-alone' programming unit, in the sense that its internal details are hidden from everything outside the module.

Intermediate level definition comes next. All of the levels between the top and the bottom levels are defined and decomposed into modules.

Stages 1 to 3 are particularly important, say the authors, for three main reasons. For one thing, the results of these stages define what 'correctness' means for the system—*what* the system must do. It is against this definition that the later stages will be verified for correctness. Also, the documentation of these stages provides an early, understandable explanation of the system—and a good opportunity for detecting design flaws. Finally, if these stages are done properly (that is, if they result in a good choice of abstract machines and modules), then the following stages will probably turn out to be simple, even for complex systems, according to the authors.

The representation stages. Representation consists of two stages which probably represent the 'technical heart' of the methodology.

Module specification is reasonably complex. We will be able to treat only a few of the highlights.

The HDM specifictions have been designed to have three significant properties, say the authors. They can be read and understood by a diverse group of people, to aid in inspections and critiques. They are machine processable, so as to provide for automatic checking for syntax and other types of errors. And, finally, they need not be compilable; rather, they describe the functional behavior of a collection of programs that implement a module.

Types of expressions. There are five main types of expressions through which the functions of a module are described. One, of course, is arithmetic and another is conditional ('if...then...'). Then there is a simple boolean type of expression, the values of which are either true or false. Next, a relational expression may be constructed from two simple boolean expressions, such that if one is true, then the other must be true. The final type of expression is 'quantified,' to express properties relating to a large number of values. An example of a quantified expression is, "For all values of X such that P(X) is true, then Q(X) is also true."

Functions. The functions that a module performs are of three basic types: (a) one type returns a value to a requesting function, but does not change the state of anything within the module, (b) another type changes the state of something within the module but does not return a value, and (c) in the third type, both things occur—a value is returned and a state is changed.

An example of the first type, where a value is returned but no state is changed, is the answer to the question, "What is the next available buffer location?" Nothing within the buffer is changed, and the answer can be the location or can be 'none,' if the buffer is full.

The other two types of functions involve a change in the state of something within the module. Since state-changing is critical, this is where the 'modes of failure' analysis plays a key role. *All* exceptions should be identified and the consequent actions to be taken defined. In HDM, each exception is named and is given a boolean definition of the condition under which it is true.

A state-changing operation (of a module) thus either performs the requested state changing and returns the 'normal' value, if appropriate, or else it returns one of the exceptions. Each type of exception is checked, in turn, to see if it is true. If an exception is true, the operation returns the value of that exception (such as 'buffer full'). If none of the exceptions are true, then the normal value is returned, such as filling the next buffer location and providing the pointer for that location.

It bears repeating that SRI developed the SPECIAL language for specifying modules in these terms.

Data representation. Somewhat in parallel with the development of the specifictions for the modules of an abstract machine, the data structures that will be used by that abstract machine must be defined. These definitions are made in terms of the data structure of the next lower level machine, say the authors.

As we see it, an example might be the following. Consider a payroll application where one level of abstract machine might be titled 'compute pay.' This abstract machine might consist of two modules—'compute gross pay' and 'compute net pay.' At the next lower level, 'compute gross pay' would be considered to be an abstract machine that might consist of four modules for computing regular pay, overtime pay, sick pay, and vacation pay. At the higher level abstract machine ('compute pay'), the data might be specified only as 'gross pay data segment,' without giving attention to what makes up that segment. But at the lower level ('compute gross pay'), it becomes clear to the designer that data will be needed for regular hours worked, overtime hours, sick time, and vacation time.

These, then—module specification and data representation—are the stages that make up the representation phase of the development project. The next stages are implementation.

Implementation stages. It is in the implementation stages that the use of SPECIAL ceases. Instead, a form of pseudo-coding is first used, after which the programming language in which the modules are to be written is used.

Abstract implementation (of the operations of each module within an abstract machine) is performed by writing an abstract program for each module within the abstract machine. Each program 'calls' on lower level modules, as appropriate.

SRI has developed ILPL (intermediate level programming language) for writing this level of code. While other languages could be used for this purpose, ILPL has the advantage of being compatible with the syntax, type checking, and other parts of HDM, say the authors.

In use, we gather, the programmer first writes an 'informal' set of code for a module, using natural language sentences—an informal pseudo-code. Statement types include 'retrieve,' 'modify,' and 'if,' and describe what the module is to do. When this has been done to the programmer's satisfaction, it is converted to ILPL, a formal pseudo-code.

The ILPL code is amenable to a variety of types of automated checking, for detecting errors. The reason for using ILPL is, we gather, that it is very easy to write from the informal pseudo-code, since the two involve the same level of detail. But once in ILPL, the code can be automatically checked, for flushing out errors, before the regular coding begins.

Coding. When the tools are available, the ILPL code can be automatically converted into regular code of the programming language that is being used. In the absence of such translation tools, this code conversion can be done manually. Since the module design has been established and most of the errors removed, this coding tends to proceed very rapidly, we are told.

Verification. As mentioned earlier, formal verification under HDM takes place during the conceptualization, module specification, and coding stages. At SRI, they have performed such verification at all three stages. For instance, they have performed the design proof, are engaged in doing module code proofs for selected modules, and have a module verfication system in operation, for the KSOS operating system. They also are starting on proofs of Pascal code for a high reliability airplane control system, for which some design proofs have been done. So verification methods are at least available for experimental approaches to 'real-world' systems.

This, then, is a brief overview of HDM. It has been designed to support the creation of hardware/software systems that exhibit very high integrity. As we have attempted to point out in this report, higher levels of system integrity are needed than are found in many of today's business data processing applications. System development methodologies of the future are almost sure to stress this need and their ways of meeting it. HDM provides a glimpse of how future methodologies will approach high integrity.

But a word of caution is in order. HDM is still under development; it is not yet ready for everyday use. When it *is* ready for the marketplace, chances are that learning to use it will be a real challenge for many system analysts and designers. Of course, it may spawn less demanding methodologies that can be used when less-than-extreme levels of system integrity apply.

If you hear complaints that "new computer systems still do not do what they are supposed to," HDM illustrates how difficult it really is to achieve good system integrity.

REFERENCES

1. Reports by SRI International (333 Ravenswood Avenue, Menlo Park, Calif. 94025), prepared by the SRI Computer Science Laboratory:
   a) Robinson, L., "HDM—Command and staff overview," Technical Report CSL-49, February 1978.
   b) Roubine, O. and L. Robinson, "SPECIAL reference manual," Technical Report CSG-45, January 1977.
2. *Compcon Spring 79, Digest of Papers*, IEEE Computer Society (5855 Naples Plaza, Suite 301, Long Beach, Calif. 90803), a series of papers on "the state of the art in computer security technology," pages 29-43; price $20.
3. *Proceedings of the National Computer Conference*, AFIPS Press 1815 North Lynn Street, Arlington, Virginia 22209; price $60 each:
   a) Neumann, P., "Computer system security evaluation," 1978 Proceedings, p. 1087-1095.
   b) Feiertag, R. J. and P. Neumann, "The foundations of a provably secure operating system (PSOS)," 1979 Proceedings, p. 115-120.
4. Levitt, K. N., P. G. Neumann, and L. Robinson, *The SRI Hierarchical Development Methodology (HDM) and its application to the development of secure software.* This report is being published by the U.S. National Bureau of Standards "in late 1979." For information on ordering, write Systems and Software Division, NBS, Washington, D. C. 20234.