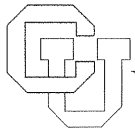
# Software Process Interpretation and Software Environments *

**Leon Osterweil**

**CU-CS-324-86**



**University of Colorado at Boulder**
**DEPARTMENT OF COMPUTER SCIENCE**

• This work was supported by grants from the U. S. Department of Energy numbered DE-FG02-84ER13283 and from the National Science Foundation numbered DCR-8403341.

# Software Process Interpretation and Software Environments

by

Leon Osterweil

CU-CS-324-86 April, 1986

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309-0430
USA

This work was supported by grants from the U.S. Department of Energy numbered DE-FG02-84ER13283 and from the National Science Foundation numbered DCR-8403341.

## ABSTRACT

This paper suggests and explores the idea that software engineering processes should be specified by means of rigorous algorithmic descriptions. The paper suggests that rigorous encodings of such processes as development and maintenance, can serve both as guides to better understanding of the processes and as a guides to superior architectures for integrating software tools into environments.

The paper presents some algorithmic specifications for some key software processes and uses them to show how examination and elaboration of these process descriptions leads to deeper understandings of the processes, better appreciation of the relations of the processes to each other and to a truer understanding of such software engineering notions as reuse, metrics and modularity.

The paper also suggests that software process encodings should themselves be viewed as items of software which have been created as products of development processes. Examination of such process development processes leads to further insights. The paper also suggests that maintenance should be viewed as an activity aimed at the modification of software objects, where these objects may equally well be applications software or software processes. This view leads to a satisfyingly general and powerful architecture for the integration of diverse software tools into a software environment.

The paper also addresses the question of what sort of language might be needed in order to do an effective job of supporting the expression of software process programs.

## 1. INTRODUCTION.

The goal of software engineering is to systematize the processes of developing and maintaining software, thereby improving the quality of the end product and reducing costs. A great deal of research has been aimed at attempting to understand how software should be developed and maintained (eg. through methodology studies), how improvements in current practice can be effected (eg. through the application of automated tools and aids), how well various tools and methologies are working (eg. through the creation, application and refinement of software metrics), and how the entire software process can be qualitatively changed and improved (eg. by treating it as a manufacturing process which exploits reuse of major components). The approach presented here incorporates some elements of all of these activities and should stimulate all by showing the mutually enriching relations that each bears to the others.

This paper suggests that software engineering must become the study of how to formalize, specify and automate the complex processes by which software products are synthesized out of very large and intricately structured collections of software objects. We suggest that software systems are most profitably thought of as products of uncomfortably large and complex manufacturing and fabrication processes, but that these might well be very effectively described, automated and controlled by surprisingly comfortable and familiar algorithmic programming techniques.

This suggestion should not be too surprising, as software is a product which is built entirely out of data (as opposed to other products which are merely modelled by data). Thus it seems only logical that the effective application of computing power should be a promising and appropriate medium for control of its manufacture and management. Thus this paper suggests that the large and intricate object which is a software product should be viewed as a data processing product to be created and managed with computing resources and software.

This view obliges us to take a fresh look at the processes which surround the development and maintenance of this product. This suggestion then leads to a gratifyingly elegant unifying perspective on such key software engineering issues as reuse, maintenance, quality assurance, software metrics and lifecycle models.

It also suggests how these processes can be used to guide the application of software tools to support the processes, and how these tools might be effectively integrated into environments to automate the processes.

## 2. BACKGROUND.

One of the longest standing goals of software engineering has been the effective application of computing power to the task of assisting the development and maintenance of software. The earliest efforts at doing this resulted in the creation of individual computer programs, called software tools, which provided computer support for carrying out various software tasks. Software tools have been produced in considerable numbers and variety over the past ten or so years. Few have been widely used to good effect, however.

Currently, it is popular to suggest that tools have been unsuccessful because they have focussed on restricted aspects of larger software development processes, rather than on entire processes. It is suggested that collections of tools artfully integrated in support of entire processes will succeed where individual tools failed. Thus efforts at supporting and automating software have shifted to focus on environments--integrated collections of tools. Environment research has attempted to determine satisfactory rationales and strategies for integrating software tools in such a way as to increase their impact. Here we suggest that effective integration of tools into environments is tantamount to codifying and automating well defined software processes. We can start to appreciate this by studying the history of successes and failures in tool and environment utilization.

It has been a disappointment and an apparent paradox that some of the most sophisticated and powerful tools have seen the least use. On the other hand, it has been observed that some of the most effective tools are ones which focus most sharply on smaller tasks and which, therefore, are able to strongly support those tasks directly. It has been suggested that the failure of the larger and more powerful tools is that the facilities they have offered have not been well matched to the tasks in need of support, and that they have not been well fitted into the larger processes of which these tasks are themselves a part.

It has been suggested that the main problem with larger tools is that they attempt to support collections of smaller tasks and that these collections of tasks are organized into processes in different ways by different software organizations. Thus it is correspondingly difficult or impossible to create a single large tool which will be well enough matched to the variety of extant processes to be widely adopted and widely effective. This realization has caused software tool makers to become increasingly interested in the processes by which software is developed and maintained in actual practice. The hope is that software toolsmiths might then be able to provide effective support for these actual processes, while also coming to understand needed improvements to these processes and laying the groundwork for evolutionary migration to tool support for these improved processes.

Environments are perhaps best viewed as systems for supporting entire software processes. Experiences with software environments have also indicated that environments which are most sharply aimed at supporting specific software methodologies are most successful and most effective. The very high costs of producing software tools and environments, however, serve to discourage the idea of producing separate sharply aimed specific software environments for different software processes and activities. Instead it has been suggested that tools and environments should be generated out of metatools and metaenvironments (eg. [Reps 85] [Gandalf 85]) or that they should be tailored or adapted out of flexible and/or extensible toolsets (eg. [Osterweil 83]). In both cases the aim is the same--the production of a comprehensive, smoothly integrated collection of software support tools and aids capable of providing strong, sharp support for people engaged in the various activities comprising some specific software process (eg. development of new software or maintenance of existing software). In both cases there are some very important problems to be solved.

Perhaps the most fundamental obstacle to being able to create sharply tailored, highly effective toolsets and environments is the very primitive state of our ability to understand and specify the processes which these toolsets and environments must support. There has been some significant work in studying, comparing and evaluating the way in which certain specific software tasks are carried out. There have been attempts to understand and describe how design is and should be carried out (eg. [JeffTPA 81] [Kant 85] [Riddle 85]). There have also been attempts to understand and describe how testing and evaluation should be carried out (eg. [Howden 85] [Osterweil 82]). These efforts have only been partially satisfactory, as they have inevitably led to an understanding that these individual activities require important linkages to other software activities if they are to be done properly and effectively.

This has encouraged study of the ways in which the larger software processes such as development and maintenance are carried out as well. Early work in this area seems to have centered on discussions of the so-called "Waterfall Model" of software development [Boehm MU 75] [Boehm 81] and on the Configuration Managment/Control approach to software development and maintenance [BryChaSie 81] [Huff 81] [Bersoff 84]. These earlier discussions had the beneficial effect of indicating that these subjects are difficult and important. There is some sentiment that study of these processes may well be the central issue of software engineering. This has led to organized and serious efforts to describe and understand the exact nature of the software development and maintenance processes. The most visible efforts in this area have centered on a recently inaugurated series of workshops aimed at understanding and systematizing the "Software Process" [SPW1 84] [SPW2 85].

We believe that the careful and rigorous description of these key software

engineering processes is critically important for a variety of reasons. Our original interest in it is to provide guidance in the development of supportive tool environments. We now also believe that this work will inevitably lead to better understanding of the underlying processes themselves, leading to improvements in the processes and the ability of software practitioners to carry them out. Our early pursuit of this research has also indicated that careful examination of detailed descriptions of the development and maintenance processes also leads to a deeper understanding of software reuse, software metrics, and software evaluation and to an understanding of the close relations of these activities to each other.

Our approach to the problem of improving our understanding and descriptions of software processes has been to describe them with algorithmic programming languages. This approach leads to some elegant characterizations of past work in this area as well as some clear indications of how significant improvements in this past work can be achieved. Thus, for example, our perspective tells us that the Waterfall Model of software development is an attempt to describe this process as a very staightforward one consisting of a strictly linear sequence of development phases. From this perspective, the early critics of the waterfall process were essentially arguing that the development process cannot be accurately modelled so simply, and were suggesting the use of more powerful and diverse control flow constructs such as iteration and branching. Advocates of iterative refinement suggested the need for procedure invocation capabilities as well.

In this paper we suggest that further control flow capabilities are needed in order to effectively portray software development. We also suggest that facilities for adequately describing the data objects to be manipulated are also needed and that adequate mechanisms for this have to this point been underinvestigated. This focus on data objects also fits nicely with growing interest in the power and importance of object orientation in programming languages and systems. In software engineering systems such as environments there have been some noteworthy attempts to portray an environment as a manager of software objects. The Smalltalk [Goldberg 84] project pioneered this approach, and subsequent work such as [Balzer GN84] has also adopted it. Our own attempts to exploit object orientation as an organizing strategy for a software environment [ClemmOst 86] have served to convice us of the power and value of this approach also.

Our experience has suggested to us, however, that the proper role of software objects in software environments is to serve as operands to software tools acting as software operators within a software engineering language which is then used to express software engineering processes (eg. development and maintenance) algorithmically.

Research exploring this premise has already begun, and has already led to a better understanding of software processes and the way in which software environments can be made more effective by supporting these processes. In addition, this work has led unexpectedly to an understanding of a variety of other software engineering issues, such as the relation of software reuse to the maintenance process, and the place and purpose of software metrics such as those described in [Basili 80] [KafuHenr 81] and [McCabe 76] in supporting the maintenance of software processes themselves. In short, we have discovered that this research premise leads to a deeper understanding of the nature of software engineering and to a better appreciation of the relations between various of its major problems.

In the following sections we elaborate upon these ideas and present more details of this research.

## 3. PROGRAMMING SOFTWARE ENGINEERING PROCESSES.

In this section we provide concrete illustrations of what we mean when we talk about "programming software engineering processes." We develop some algorithmic descriptions of some well-known software engineering activities. Our basic approach is to use iterative refinement to go from a conceptual and structural view of the processes, to the lower levels of detail at which the important specifics of the various subphases and subprocesses can be seen. As we delve into the lower levels of detail we gain insights into the processes themselves, other (sometimes unexpectedly) related processes, and the tool supports which they all need. We discover that in order to effectively describe software engineering processes we require unexpectedly powerful algorithmic constructs and data description and definition capabilities. We also discover that the proper treatment of testing and evaluation seems quite hard, and may perhaps entail the use of different programming capabilities. As we proceed with investigations of the software development and maintenance processes, we are drawn into deeper considerations which lead to improved understanding of other processes, some of which may not have seem related at first.

### 3.1. THE SOFTWARE DEVELOPMENT PROCESS.

We begin by constructing a sequence of algorithms describing the software development process. The sequence begins with a very high level algorithmic description expressed in terms of very high level data objects and correspondingly high level processes. Thus, at this level we consider a body of source text to be a single object. Other objects at this level are testcase sets, requirements specifications, executable programs and test results. At this level the corresponding operations include compilation, test-execution and design

synthesis.

The first algorithms are tantamount to somewhat formalized renderings of the development process described by the Waterfall Model and some subsequent enhancements to it. At lower levels we uncover details which suggest algorithms which seem to us to be more satisfying descriptions of software development than we have seen suggested before.

To start, the highest level algorithm depicting the classical view of program development might look something like Figure 1. This program abstracts away from view virtually all of the substantive issues, which are concealed in such main procedures as create_design, fix_requirements, design_ok, and fix_code. Interesting and important features appear as we begin to elaborate on these details. We could choose to elaborate upon such major subprocedures as "create_design", but shall not do so here. Active work in the area of studying design (eg. [Kant 85]) and in comparing methodological approaches (eg. [Riddle 85]) should lead to understandings enabling us to provide sharper detail of this code. Conversely, we also believe that the approach we are suggesting here will be of considerable value in helping those researchers to sharpen and rigorize their discourse and focus.

Instead, we choose to elaborate upon some details of the testing and evaluation subprocesses. This is an area in which we have ongoing research interests (eg. [Osterweil 81] [Osterweil 82], and is also an aspect of the development process whose investigation and elaboration seems to lead most rapidly to substantive and productive new views.

### 3.1.1. AN ALGORITHMIC VIEW OF TESTING AND EVALUATION.

We begin by attempting to focus on the procedure "code_ok". This procedure has as its job determining that the code which has been created is an object which is satisfactory in the context of the other software objects. In earlier papers we have discussed the difficulty of deciding when a body of code should be considered to be satisfactory [Osterweil 81, Osterweil 82]. In those papers we expressed the view that the central issue in such considerations must be the presence or absence of errors, and that the notion of "error" is a relative one, with an error being a deviation of the program's actual behavior from its expected behavior.

Thus we have previously expressed the opinion that the testing and evaluation of code should be a question of determining its consistency with specifications. In this section we examine more closely the ramifications of this viewpoint in the context of algorithmically expressing the software development process.

We are led to conclude that our procedure "code_ok" must consist of code to verify various consistency relations, among various software development objects. Decisions about just which consistency relations are to be verified by these procedures are squarely the responsibility of the software project management. On the other hand, management should be in a position to select these procedures from among a body of available procedures and techniques, each of which is described by algorithmic code and supported by software tool configurations. The way in which these various procedures and techniques might be embodied, selected from among, and evaluated by testing and evaluation practitioners is a difficult and crucial issue, which is a central concern of this paper. In order to address it most effectively, however, we must defer it until our basic ideas have been more fully and carefully presented.

Thus, for the purposes of this discussion, suppose that it has been decided that "code_ok" is to be considered true if and only if the execution of a body of test cases produces results that are identical with prespecified results. In order to write code for this, declarations such as those shown in Figure 2 are necessary. With these declarations, it is now possible to code the algorithmic portion of "code_ok". This algorithmic specification is shown in Figure 3.

Figure 3 has several noteworthy features. One feature worth pursuing is the procedure "derive" which "code_ok" invokes. The purpose of this procedure is to create the object named as the first argument using as input the software object named as the second argument. The effect of execution of this procedure is not merely the creation of the new object, but also the establishment of a "derivation" relation between input and output objects. We believe that this "derivation" relation is a very fundamental technique for organizing and structuring the large number and variety of software objects needed to comprise a software product. In this example the derivation process is intended to be carried out in both cases by automated tools, but that need not always be the case. We shall return to discussion of derivation later in this paper.

Another feature of the previous example is the procedure "consistent." This procedure is used to determine when and whether a test result is "consistent" with expected results. In some cases this procedure may be elaborated to something as simple as determining the equality of two integers or the bit-for-bit equality of two files. In other cases, complex numerical analysis may be required in order to assure that test results are "consistent" with specifications. In any case, however, such details can and should be supplied in lower level procedures. These lower level procedures should be created and placed in a library by testing experts who should also, ideally, create automated tools to support the procedures. This having been done, the selection from among these libraries, and assimilation of selected procedures and support tools into "code_ok" should be done by software engineers and project management in

accordance with project needs and guidelines. The details of how this can be done is presented in subsequent sections of this paper.

Some difficulties with the procedure in Figure 3 cannot be solved by simple iterative refinement, however. There is a major architectural weakness in the procedure which appears as one considers how the lower level subprocedures of "code_ok" must interact with each other. In actual practice, one rarely waits until all code has been written before embarking on a monolithic testing regimen. In practice, code is written in small increments, and each increment is tested progressively carefully. At first testing may be difficult or impossible because only small pieces of the code exist (ie. the procedure derive (code.executable, code) is itself unexecutable because the hierarchical structure, "code", is still incompletely defined). In this case, practitioners carry out such procedures as checkout compilation to seek syntax and elementary semantic errors, and test execution using test harnesses and testing stubs.

Thus, the major difficulty with "code_ok" is that, if it is to correctly represent the way code testing is and should be carried out, it will not fit neatly into the linear algorithmic flow of the main procedure develop_software. This should not be at all surprising as most observers have concluded that software is virtually never built in the neat linear fashion shown there, which we now understand corresponds to the most simplistic algorithmic encoding of the "waterfall" development process. Our current example indicates that once coding begins, there are also testing activities interspersed with it, that these may necessitate peripheral coding activities (eg. to create test harnesses), that they may entail revisitation of requirements specifications (as testcases are devised), that they may uncover inconsistencies either within the evolving code object (necessitating alterations to the code objects) or between the code object and the requirements and/or design object(s) (necessitating changes to any or all of the three).

Other observers have previously lamented this apparently chaotic situation. Nevertheless it has also been observed that this apparently chaotic situation virtually always prevails in actual software development practice, and that it is sometimes successful in producing an acceptable software product. Thus the situation is apparently controllable, and in fact ought to be expressed algorithmically. In doing so it them becomes possible to to show how the apparently chaotic process is controllable, thereby enabling more effective control, and also indicating how tools can be used to effectively support the process.

### 3.1.2. DEVELOPEMENT AS AN OBJECT CREATION/ASSIMILATION PROCESS.

We start by observing that the ostensibly chaotic situation just described is in fact not all that chaotic, but can and should be viewed as the progressive process of creating new code objects and meticulously incorporating them into the large and evolving structure being created by the development process. One of the key ways in which such new objects must be incorporated is by assuring that they are connected by all of the "derivation" and "consistent" relations which are created as directed by the algorithmic code of the "develop_software" procedure. When inconsistencies are encountered, either the new objects must be changed or withdrawn, or older objects must be changed or withdrawn. Obviously this latter can entail considerable trauma as the older objects should already be connected to other older objects by previously established relations. Nevertheless, this seems to be an accurate view of how a code object is developed. It suggests that the algorithmic process for developing code should be as shown in Figure 4.

Here again we must delve into the details of such procedures as ok() and change_design() in order to appreciate the subtleties of the higher level procedure. We shall focus on procedure "ok()" which is designed to determine the acceptability of a newly created piece of code. Algorithmic code for this procedure might look something like the algorithm shown in Figure 5. It is important to observe that this procedure is generally executed during code development in addition to, rather than instead of, procedure "code_ok" which was previously presented.

Clearly the procedure for determining whether or not the code structure S is complete is likely to be a complicated procedure probably requiring considerable static analysis. On the other hand, the existence of such a procedure, reliably implemented by software tools, can be invaluable to software developers in advising them as to when and whether test harnesses need to be contrived because existing source is insufficiently complete to support testing activities.

It is also clear that the procedure build_test_harness is likely to be a very complicated one which will have to incorporate subfunctions for accessing design specifications for invoked (but currently uncoded) subprocedures, for recapitulating the entire development process in order to construct code for the test harness, and (probably) for deciding whether or not the work entailed in doing all of this is warranted. In fact, it is likely that "build_test_harness" must be a Boolean procedure which returns FALSE when it is decided that the effort of building a test harness is not currently warranted. In this case the subsequent testing procedure will have to be a null procedure, thereby leaving undone the process of determining whether certain of the software objects being developed are sufficiently "consistent." Thus this decision is one which often has far reaching consequences. If it is encoded in the context of a precise development algorithm it can be based upon accurate information and reliable estimates.

Once made, its consequences can be made clear in terms of software objects and consistency relations omitted.

We should not need to explicitly show the algorithm for implementing the procedure "change_design ()" in order to make the point that the implementation of this procedure will necessarily entail the modification of some subobject or subobjects contained within the design object at the cost of considerable trauma. As that object has presumably been related to numerous other objects in order for the process create_design and the process ok_design to have completed successfully, any alteration to the subobjects within the design object will necessarily occasion the need for the reconsideration and reevaluation of all of these relations. The ripple effect of such a change has been noticed in all development projects. The usual problem is that the extent of the ripples is difficult to determine accurately once changes begin to cascade. Here the use of a formalism would seem to have considerable value. Ripples continue exactly as far as previously defined consistency relations continue to be evaluated to the value FALSE. As the consistency relations have been previously defined and the objects on which they must be imposed have been previously created and placed within a structure, there should not be any difficulty in deciding when the effects of changes must be propagated further and when they need not.

The foregoing examples illustrate the point that software development can and probably should be thought of as the process of creating new objects, determining the extent to which they are consistent with previously defined mandated consistency relations, making alterations to either the newcomers or previously created objects, rippling the effects of these changes outward, and continuing with the process of creating new objects.

This view has certain satisfying aspects, some of which have been touched upon before. In some ways, it continues to present problems, however. One problem is that is does not appear to provide much guidance as to the order in which new objects are to be created, nor the choice about when to evaluate consistency relations and when (and how) to ripple out the effects of changes.

### 3.1.3. A MODULAR SOFTWARE DEVELOPMENT PROCESS.

One strategy that seems to make a great deal of sense is to create software objects in clumps, where a clump consists of a set of objects which must be tightly interrelated within the larger structured software product. This strategy enables the software developer to create closely related objects either in parallel or in close time sequence. This makes it easier to evaluate the needed "derives" and "consistent" relations which must bind them and to be more sure that these relations will be satisfied.

What is being suggested here is not too dissimilar to the notion advanced by researchers (eg. [Parnas 72], [LiskZill 75]) advocating the use of modules. They suggest that a module is a tightly interconnected collection of software objects (in their case the objects are assumed to be code objects) and they are to be clustered together in order to minimize their interactions with other objects, while maximizing the chance that they will deal consistently with internally shared data structures.

Thus, from our perspective, the importance of the notion of modularity is that it advocates the identification of software objects which are most tightly interconnected and suggests that they be developed and interrelated together. It is then easier to weave them into the larger structured software product as an aggregate than it would have been to attempt to do this as a sequence of individual objects. Our view of a module is different from the classical view, however, in that we believe that a module is any tightly interconnected group of software objects, and that they need not all (or any) be code objects. Thus we believe that a module might well consist of some design objects, some specification objects and some code objects. Testcase objects might also be included in a module. This suggests that it might be advantageous to develop an entire "thread" of software objects, spanning specification through testing, more or less at once and then integrate this internally consistent collection of objects after it has been fully developed. It is interesting to observe that this is a paradigm that is often followed in actual software development projects and that it often works very successfully. This perspective might help explain why.

### 3.1.4. IMPLICATIONS FOR A SOFTWARE REUSE PROCESS.

This discussion seems to also lead to some important insights into software reuse. We should observe that software objects cannot successfully be reused until and unless they are correctly interrelated with other software objects which are being developed by the same development process. Thus reuse entails taking a product which has presumably been produced by one development process and integrating it into the product of another (possibly different) development process. This reuse activity has the greatest chance of being successful and efficient when the source and target development processes are the same or are described by algorithms which are very similar. In these cases it can be expected that many or all of the "derive" and "consistent" relations which will need to be evaluated to assure acceptable assimilation into the target product will have been evaluated in the context of the source product. In any case, if the object(s) to be reused must satisfy numerous and complicated consistency relations, the effort is unlikely to prove worthwhile. If, on the other hand, a modular clump of objects is to be reused, the "derive" and "consistent" relations to be satisfied will be fewer and simpler, and the reused software objects will be

Thus it appears that in order to successfully program the software development process, the algorithmic language used must support the expression of parallel tasking, access to shared data objects, and task synchronization. With such facilities it is possible to program daemons which would then be able to assess the status of the evolving software product, establish some consistency relations automatically, derive various software objects concurrently with human development efforts, enforce project guidelines, and enforce such management decisions as the freezing of development in certain areas.

Using such an algorithmic language the procedures create_code, create_design and create_requirements would be implemented as tasks and software practitioners might participate in the execution of several of these tasks concurrently. Procedures such as code_ok would also execute concurrently, but they might be executed, at least partially, under the control of daemons. The specification, design and coding of such daemons seems to be an important research problems in software engineering. We believe that we are, in this paper, creating a setting and structure within which these tasks can be readily understood and within which the needed procedural code can be expressed and experimented upon.

### 3.2. PROGRAMMING THE MAINTENANCE PROCESS.

In this section we explore the notion that software maintenance is the process of altering the form of a software product, the contents of a software product, or the process by which such products are produced (or any combination of all three). We begin by observing that all of these types of alteration are currently done quite routinely but under the single heading of "software maintenance." In failing to recognize that they are distinct, however, and that they have differing ramifications, they are often carried out incorrectly with consequences which are often expensive and disastrous.

One way to help distinguish among these three types of maintenance is to think of any particular software product as a template filled with information. Any particular software product is profitably and accurately viewed as one such filled template, which is built up by aggregation of the of more primitive subtemplates and objects which comprise the final software product. Such a filled template is clearly quite a bit different from the template itself. Both are, in turn, quite a bit different from the process by which an empty template is filled with values, producing a completed template.

This observation suggests that there are three principal objects of importance in software--the software product, the template which structures the product and the process by which the structured product is developed. We will have much

more numerous. Among the implications of this observation is that reuse of code objects alone is likely to be ineffective, especially if these code objects are extracted from a source development product which incorporates a set of objects and relations which is significantly different from the object and relation set of the target product. On the other hand, reuse of development strands should show far more promise, especially if these strands are extracted from a development product which is similar to the target development product.

### 3.1.5. PARALLELISM IN THE DEVELOPMENT PROCESS.

Although there seems to be much promise in the previous suggestions that the development process be viewed as the incremental creation and assimilation of modules of software objects, there is a problem in that straightforward sequential development algorithms do not accomodate this sort of development stategy very well. On the other hand a development model capable of expressing asynchronous, parallel development tasks seems to accomodate it very well and show considerable promise. A reasonable software development algorithm might be one in which the development of each module of a software product is carried out as a separate development task, and the process of incorporating such modules is expressed by a different task or tasks. The incorporation task(s), focussing on the evaluation and verification of needed consistency relations might be overtly invoked by the module writer or project manager, or might be invoked by a daemon. The daemon might simply periodically check the status of the project, or it might be triggered by some prespecified set of conditions corresponding to a preconceived notion of drastic change in the status of the product.

The need to inject asynchrony into the proposed process is further motivated (strongly) by the need to accurately model how software is to be produced by teams of workers. It has often been observed that software engineering is most desperately needed to assist the development of large software by large teams of workers, and is far less important as an aid to a single software developer. In the case of team development it should be clear that each developer is, in effect, a parallel processing activity, but that all are sharing access to the central object which is the evolving software product. Thus, the development process is being carried out as a collection of asynchronous development tasks--with each worker executing at least one such task. It follows that the task of assuring the consistency of the software product under development is crucial and enormously difficult. This task falls to the software manager, most of whom will attest to its enormity. We suggest that the success of a large software development activity most often hinges on the ability of the software development manager to effectively synchronize the myriad development tasks in such a way as to preserve the consistency of the evolving product.

more to say about these and other software objects in later sections of this paper. For the purposes of this section, however, let us simply observe that the three types of maintenance just referred to correspond to processes aimed at the alteration of the three types of objects just enumerated.

The first type of maintenance entails alterations to a filled-in template. The second type of maintenance entails alteration of the template itself. The third type of maintenance entails alteration of the process by which individual instances of a filled-in template are created.

For example, sometimes it is deemed necessary or important to alter the "derives" and "consistent" relations defining the structure of the software product. This is an example of the second type of maintenance. Sometimes alterations are made to the order in which the creation and verification of such relations are carried out. This is an example of the third type of maintenance.

Either type of alteration is quite distinct from the first type of maintenance in which only specific data contents of a single software product is carried out. Thus, adding new code and/or testcase objects to a software product, is an example of the first type of maintenance. This entails the transformation of one particular filled-in template into another filled-in template. This requires that new objects be created and inserted into the existing template. This activity would presumably be carried out when the executable code object contained in a developed software product was found to commit errors or be incompletely tested.

Maintenance aimed at altering the template itself is a more complex and significant maintenance activity. It is required when it is discovered or decided that the product structure which is which has previously been used to guide the development of past software products itself is either faulty or incomplete and has to be changed. Under these circumstances it is necessary to change the structure itself, begin using it to develop all subsequent software products, and somehow go back and transform earlier products, which had been created by essentially filling with objects the template which has now been changed. Later in this paper we shall discuss how the need to change the template itself may arise and be recognized.

Whatever the reason for carrying out the maintenance activity, in this section of the paper we are concerned with indicating how such an activity can be expressed algorithmically. To begin, we address the question of how one filled-in template can be transformed into another such filled-in template. An algorithm for receiving a request for such a change, and carrying it out would, at the highest level, look something like the procedure shown in Figure 6.

It is important to address the problem of deciding how and when to attempt to reestablish the consistency relations involving altered objects. Clearly one could elect to include at the end of each of the "change_" procedures the invocation of a consistency checking procedure which would then be responsible for verifying the soundness of the software product after each change. As a change may involve the alteration of more than one object, however, this raises the question of when the consistency checker ought to run--after each object has been changed, after several have been changed, or after all have been changed?

Compounding this difficulty is the fact that maintenance activities on actual software systems are generally carried out in parallel. In a commercial software operation the maintenance organization generally acts as a funnel, receiving requests for maintenance and then dispatching them to individuals for remediation. From the point of view of simplicity and control it would be easiest if this dispatching procedure were carried out as a sequential loop such as shown in Figure 7. This is quite different from the way maintenance is actually done however, as the actual process involves carrying out multiple maintenance activities in parallel, adjusting the priorities given to various maintenance activities, dynamically deleting some, and so forth. Thus, the dispatching and handling of maintenance jobs is generally carried out as a multitasking activity. Further, our previous discussion has already suggested that the process of checking on the soundness of the result of a maintenance activity is best thought of as a task which may not always be started under the direct control of the maintainer.

In practice the responsibility for checking the consistency of a change with preexisting software objects and structure is often carried out by a special task (the "configuration manager"--often a person or committee). We now suggest that consistency checking might be the responsibility of a daemon which is started up either by a timer, or by other daemons whose jobs are simply to monitor the flow of alterations to the software product. In any case, we believe it is noteworthy that it is far more satisfactory to attempt to express the way in which maintenance is and should be done in terms of concurrent tasks than by means of algorithms which are not able to exploit such control flow primitives.

Similarly this seems to be an appropriate mechanism for helping to control the larger and more complex process of transforming the template used to define a software product into an another template, corresponding to a software product structure. As observed above, this sort of maintenance also entails the transformation of existing filled-in templates into instances of the new template, where previously created objects may have to be relocated, deleted, transformed or related to different objects, perhaps in different ways.

The transformation of these old filled-in templates into new templates might

well be carried out as described above once the new has template been defined and put into effect. The process of setting up the new template, will be a different process, however, aimed at the transformation of a different sort of object. The process of "publishing" the new template and "putting it into force" is another process still. All of these processes can and should be described algorithmically, thereby comprising an algorithmic picture of what this second type of maintenance entails.

### 3.3. DATA OBJECTS IN SOFTWARE ENGINEERING PROCESSES

Although the preceding discussions have been primarily aimed at furnishing algorithmic details of the development and maintenance processes they have also strongly indicated that a great deal of data object description is necessary in order to specify these algorithmic details. This should not be surprising as both software developers and programming language designers have become increasingly impressed with the importance of powerful and precise data specification in creating superior programs. Thus, as we are suggesting a programming approach to describing software processes, it seems only natural that we should devote considerable attention to defining data objects and structures for sharpening and facilitating our algorithms. The alert reader has probably noticed that data object issues have already entered some of our earlier discussions. It is now time to address them directly and systematically.

#### 3.3.1. STRUCTURING THE OBJECTS OF A SOFTWARE PRODUCT.

Our previous presentation of software process algorithms have most noticeably required that some structure be placed upon the diversity of objects upon which they were operating. In "code_ok", for example, testing was viewed as an activity which processed an array of testcases. The testcases, in turn, consisted of pairs--test inputs and specifications of required outputs. Our subsequent discussion of the difficulties presented by the need to test code while it is still under development touched upon the fact that code should be developed in pieces and integrated incrementally. This left unstated, but nevertheless quite obvious, the fact that a code object must be viewed as a hierarchical structure of lower level subobjects, modules and compilation units. Thus we have already seen instances of the need to place a hierarchical and aggregation structure upon some of the software objects manipulated by the algorithms we are developing.

We also emphasized our conviction that testing and evaluation are processes which tantamount to the establishment of a different sort of structure--namely a "consistency" structure. A consistency structure of software objects relates to each other those objects whose contents must be be consistent in certain

specified ways. The purpose of this structure is to define what constitutes an error in the software product being built and to specify ways in which these inconsistencies (errors) can be detected.

Still another type of structural relation that our previous discussions indicated was the "derive" relation. This was mentioned briefly in the context of our discussion of testing, where it was indicated that an executable program object was to be created out of an existing source text object. There we observed that the new object was not simply created as a freestanding entity, but rather that it was created and related to its predecessor object--the source text--by a derivation relation. We believe that it is important to make such derivation relations visible and explicit as an aid to software practitioners and managers. In addition, in subsequent sections of this paper we shall see that making this relation explicit can be used to greatly improve the efficiency of automated tools and integrated environments.

In our early investigations we have been surprised at the number and complexity of the relations which seem necessary in order to express the structure of a software product. We have found that a given object is generally involved either directly or indirectly (eg. through derivation chains) in several different hierarchy relations, several different consistency relations and several different derivation relations. In addition, the structures of these relations need not be related to each other in any particular way. Thus, a given object, say a modular group of source text objects, may be hierarchically related to a particular set of objects (eg. if it consists of a set of separate compilation units, and is also included in a set of higher level procedures), but may be a part of some different "consistency" relations (eg. if it is necessary for it to be consistent with some project guidelines concerning source text size, structure or format), and may be a participant in still different "derivation" relations (eg. if the module is a prettyprinted or instrumented version of a different source text object).

We are left with the feeling that software products have far more structure than we had previously believed or than we had expected when we began this investigation. We note that Howden has also discovered that complex structural relations among various types of software objects, spanning several software lifecycle phases, are necessary in order to carry out testing in the way he suggests [Howden 85]. Thus we are increasingly convinced that computing support is essential in specifying what these relations ought to be and how they are to be established. Specification and establishment of the various relations is the job of the individual who creates the process which builds and exploits these relations and it should be carried out according to algorithmic steps which are part of the process.

### 3.3.2. TYPING SOFTWARE OBJECTS.

We also strongly believe in the importance of attaching type attributes to the objects of a software process. The "derives" relation encourages the view that the data objects manipulated by software processes are mostly derived from each other by the action of tools. This suggests that tools should be thought of as operators which transform objects (operands) into other objects. There seem to be a number of advantages to this view, to tool writers, process encoders and software object managers alike. Thus, for example we believe one should consider a compiler to be a transformer which maps objects of type "source_text" into objects of type "object_code". Probably any specific and actual compiler is probably best thought of as mapping a "source_code" object into an object of a type which is specific to that compiler and a subtype of type "object_code" from which it inherits certain properties and characteristics (eg. loadability). Different compilers for the same language dialect would be thought of as distinct operators tranforming "source_text" objects into different subtypes of the "object_code" type.

A specific program execution environment can also be considered to be a tool which implements a transformation operation. This operation is capable of mapping pairs, consisting of an object of type "executable_program" and an object of type "test_data" into an object of type "test_results." When important or necessary objects of these various types can and should be coerced into objects of different types-- for example when it becomes important for a "source_text" object to be used as input to some tool which may have been created to deal with arbitrary raw character strings.

Many of these notions were advanced and implemented as part of our research in creating the Odin software tool integration system [ClemmOst 86] and in using Odin to integrate some powerful tool collections. Odin is perhaps best viewed as a manager of large grained persistent software objects. It uses hierarchy and derivation (although not consistency) relations to organize, manage, and optimize the creation of these software objects.

Objects in Odin are also typed. In fact, the structure of the types operated upon by the various tools which Odin integrates is captured and expressed in a separate object--the Odin dependency graph (see [Clemm 86] [ClemmOst 86]). By creating this type structure as a separate object we discovered that it was a great deal easier to maintain the type structure (eg. by altering or adding new types and new tools). From the perspective of this paper we can now see that the Odin dependency graph is a vehicle for the definition of the structure which can be placed upon the types of the objects being manipulated by the users of the various software processes which are interpreted by Odin.

This perspective helps us to understand the nature of software development and maintenance better, and reinforces our belief that data definition and description play very central roles in effectively programming these processes. We now understand that software development is the process of defining the type of the software product(s) to be produced, then instantiating the product(s) type(s) and finally elaborating and evaluating the needed instances. This view of the development process is illustrated in Figure 8. Interestingly, this view points up the fact that most software development process attention has been focussed on creating algorithms for the elaboration and evaluation of the instances. It is clear that more attention needs to be paid to processes for defining software product types as well. Later we shall indicate, however, that this process is actually best thought of as a subprocess of the process of creating a software process, rather than creating a software product.

Earlier we also indicated that maintenance may be of three sorts. We now name them product maintenance, product structure maintenance, and process maintenance. The first is best thought of as the algorithmic process of transforming one software product type instance into another instance of the same type. The second is best thought of as the algorithmic process of transforming one type description into another type description, and then dealing with the implications that this transformation has for existing software objects and products. The third is best thought of as an algorithmic process aimed at transforming one algorithmic description into another algorithmic description. Here too this process must study and resolve the impacts of such alterations upon existing objects and processes.

Thus the three types of maintenance are all transformation processes, but are carried out on different sorts of objects. The object at which product structure maintenance is directed is a type definition, which we shall refer to as the software product structure (SPS); the object at which product maintenance is directed is an instance of such an SPS, namely a software product, which we shall denote an SP for clarity.

Thus it is clear that in order to create effective algorithmic specification of such key software processes as development and the three types of maintenance we must incorporate powerful data type specification capabilities into any software engineering language used to express these processes. We have indicated the need for the ability to type objects and object aggregates. Also critically important, however, are capabilities for attaching type specifications to the various relations which interconnect software products. We have discussed the value and importance of derivation in placing a structure on the diversity of software objects comprising a product. Now it is worthwhile to observe that "derive" is probably best viewed as a type of relation and that the various tools and agencies used in the various actual derivation processes are best viewed as subtypes

of this relation type. In this way it becomes far more natural to differentiate as needed among the various ways in which tool or other sorts of actions have developed descendant objects from ancestor objects.

The value of typing relations becomes far clearer when considering consistency relations. Our early investigations of consistency relations do not indicate clearly whether they are best expressed as algorithms, Boolean predicates, predicate calculus statements, or in some other formalism. We believe that each relation is probably best evaluated during the development process by a separate task. This serves to reinforce our belief that consistency relations should be constructed hierarchically out of lower level consistency primitives which are themselves implemented by primitive consistency checking tasks. Research must address what these primitive relations and tasks ought to be. It must also address the smooth integration of specifications of consistency relations into the software engineering language. Certainly we expect that consistency relations will be implemented in a wide variety of ways and should therefore be expected to exhibit important differences. We expect that these differences as well as equally interesting and important similarities are likely to be best identified with the help of a hierarchical type structure.

Although we have so far betrayed a prejudice towards adoption of an algorithmic programming language paradigm as the basis for the encoding of software process algorithms, the issue of adequate expression of consistency constraints serves to shake our confidence in the suitability of this paradigm. We shall discuss research issues pertaining to the selection of a language paradigm later in this paper. For now, suffice it to say that we are concerned that a language in the classical algorithmic mold may not be sufficient for the effective expression of consistency relations.

### 3.4. OTHER RELATED SOFTWARE ENGINEERING PROCESSES.

Our earlier attempts to give increasingly precise and detailed algorithmic descriptions of the software development and maintenance processes have led to deeper and more unified understanding of such other important software engineering issues as testing and evaluation, modularity and reuse. In this section we pursue these discussions further to strengthen our claim that the technique of programming these processes leads to a deeper understanding of each and the relation of each to the others. In the process we establish satisfying connections with still other software engineering activities (eg. software metrics research).

### 3.4.1. TESTING AND EVALUATION OF SOFTWARE ENGINEERING PROCESSES AND PRODUCTS.

Our suggestion that software development be viewed as a process of creating instantiating, elaborating and evaluating a data type suggests that the individual who programs this process is a classical programmer, devising data aggregates (eg. SPS's), and creating algorithmic procedures for manipulating these SPS's. We propose that this individual should be referred to as a software engineer and that individuals who participate in the creation of the actual SP's should be referred to as software practitioners or software artisans. Thus both are programmers, but the software engineer effects the development of a different, larger and more abstract object. One troubling aspect of this view is that software engineers now accept and espouse the notion that such algorithmic code should not be created until and unless other types of objects (eg. requirements and design objects) have previously been created.

Thus, as the software engineer is a programmer who creates data structures and code for the software development and maintenance processes, it is important that proper software development practice be followed in the creation of that code. The point here is that the software development procedure is indeed a piece of software itself and as such is far more than just a piece of code and associated data structure declarations (ie. the SPS). This piece of software is also a complex structure of interrelated subobjects, which include requirements, design, code, and test-related objects as well. This latter software product is the product produced by the software engineer. This product contains as one of its component software objects a process and SPS which are then used by software practitioners to guide their creation of the SP's which contain data processing systems for end users.

To lend emphasis to this dual nature of the software development process as both a process and a product of a higher level process, we shall now begin to refer to a software development (or maintenance) process as a software process-object, when we are discussing it in the context of the process by which it is itself created.

One important ramification of the observation the the software process-object is the product of a process is that we should now understand that the software process-object must in some contexts be thought of as merely an object of one type in a larger structured software product (ie. a higher level or more abstract SP) which consists of a variety of other object and relation types and instances. This perspective is an important one, because the software process-object, like all objects in any SP, must be assumed to be an object which must continue to evolve as it continues to attempt to meet needs which are constantly changing in their nature or perception or both. In the case of the software process-

object, the changes are necessary in order to assure that the process-object continues to produce data transformation systems whose characteristics are "consistent" with the expectations of those of who require that they be built.

For example, it may be the case that an executable system object, built for an end-user, must be built in such a way that no transformation ever be executed which produces data values which violate behavioral specifications laid out in a requirements specification object. If such a specification violation is subsequently observed, then the customer can legitimately question not only the executable object, but also the process by which that object was built and tested. This is because, not only did the executable object violate the customer's functional specifications, but the software development process violated the customer's specifications for the process-object.

As another example, consider the case of a software development organization in which there is a "testedness" standard in force which requires that all software development activities assure that 90% of all statements in any delivered source text object must have been exercised by a test regimen before system delivery. This requirement is actually a specification object with which all process-objects used by that software development organization must be shown to be consistent. All software engineers must devise software processes which incorporate testing procedures which must effect the 90% coverage in order to be in compliance with the organization's requirement. If, after execution of the software engineer's development process, a resulting software product does not incorporate a test result object which reflects 90% code coverage, then the software process is not in compliance with the requirement. The software engineer must either alter the testcases used or the process used in order to avert a demonstration of the non-compliance of the process-object with its requirements.

These examples are illustrated in Figure 9. They show some ways in which the need to alter a software process-object can arise, and ways in which this need can be detected. We note that detection is invariably based upon the detection of an inconsistency between the process-object and another software object. We will discuss how these other objects come into existence shortly. For now, we shall address the issue of testing and evaluation of software process-objects.

We characterize the previous two examples as testing approaches to the evaluation of a software process-object. In each case, the process-object has been created and then executed to produce a software product (SP), containing an executable data transformation system object. In each case the process-object is evaluated by studying its consistency with specifications. In the first case, we suggest that the software engineer must have specified that the development

process create a "test-result" object type, whose instances were to hold a Boolean value indicating whether or not all test cases to date had adhered to all assertions of intent. This object would then have been considered to be one of the software objects contained in the software product produced by the software engineer's software process-object. In a very real sense, the Boolean "test-result" object is an output of the software engineer's program. When this output value is False, the software engineer recognizes that his or her program is producing a data output which is inconsistent with the requirements specification object with which the software engineer's process-object must be consistent. A similar argument supports our contention that the second example is another instance of a testing approach to the evaluation of the process-object.

In each of the above cases, it seems important to note that the evaluative process is greatly improved by considering it to be a process very much like the testing processes which we currently apply to executable objects within SP's built to the specifications of an SPS. It also seems important to note that the concept of testing as being an activity in which performance is compared to expectations is not merely applicable, but also highly profitable, when applied to evaluation of process-objects. Consideration of the need to enunciate specific expectations for the functioning of a software *process,* and the separation of these from expectations for individual software *products,* seems to be a very worthwhile step. Too often it seems that these expectations have been confused, and errors in one have been approached by making changes in the other.

Thus we have seen that software process-object testing can be used to determine the effectiveness of a process in creating software products which achieve satisfactory levels of functional correctness and testedness. There should be little difficulty in imagining how process-object testing could also evaluate the success of the process-object in producing products which are satisfactorally fast, responsive, small, crash resistent and so forth.

We must hasten to observe that these testing approaches to the evaluation of a software process-object continue to suffer from the generic weaknesses of dynamic testing. Thus, they are capable of showing the existence of an instance of inconsistency between functioning and expectation, but are not able to provide assurances of the absence of such inconsistencies [Dijkstra 72]. Further, these techniques in themselves do not furnish reliability or testedness measures which are desired in order to provide managers and customers with estimates of the degree of confidence which might be placed on the object under test. Thus, we see that it is very reasonable to expect to be able to test a software process-object, but it is also very reasonable to question the adequacy of testing as the sole vehicle for gaining confidence in the process-object. It thus becomes important to consider the possibility of static analysis approaches

to evaluation as well.

Static analysis of software process-objects seems to be an eminently reasonable and realizable approach to evaluation. What is suggested here is that the rigorous algorithmic coding of software processes seems to be an excellent subject for correspondingly rigorous and effective reasoning processes about the nature and expected behavior of the processes. We believe it is important to observe that successful software development managers undoubtedly carry out such reasoning processes at least mentally to assure themselves that the processes which they are supervising are likely to be effective. Unfortunately, as these reasoning processes are invariably mental processes, studying informal or unspecified software development processes, they can be incomplete or incorrect. Certainly as the size of a software product grows, and the size of the process for creating it grows (often nonlinearly) the task of reasoning correctly about both becomes uncomfortably large. It is not difficult to point to examples of projects containing software process flaws that have interfered with the success of the software product. In such cases, the very lack of a written or rigorous representation of the process serves as a most fundamental stumbling block. Thus it would seem that our suggestion of a rigorous algorithmic expression of software processes provides the basis for an approach to this problem.

When viewed in this light, some promising forms of static analysis immediately suggest themselves. One such form of analysis is straightforward type checking. We have indicated that the objects comprising a structured software product should be thought of as being typed, and the tools and subprocesses used in creating them should be thought of as type transformers. Thus it makes a great deal of sense to analyze the software engineer's process to assure that it specifies the correct manipulation of types. This suggests the desirability of having the language in which software processes are to be programmed be a strongly typed language. It would be interesting to study the impact that making a software process language strongly typed might have on the ability of the software engineer to create effective processes.

Another promising possibility is to carry out data flow analysis of software processes [FosOst 76]. Data flow analysis is best characterized as the examination of the structure of an algorithmic procedure to determine whether or not certain specified event sequences might occur in any possible execution of the procedure [Osterweil 81]. The importance of being able to carry out such analysis of a software process is that such analysis could pinpoint bottlenecks and malcoordination in the software engineer's algorithm for creating a software product.

For example, a software engineer might have specified that a particular testing or integration analysis procedure take place at a time when the rest of the

development process may not have assured that all needed source code objects and/or test data objects had been completed. It appears that such instances of malcoordination might easily be modelled as problems of determining "liveness" or "availability" [FosOst 76]. Such analyses could pinpoint places in which the process mandated the completion of development steps whose undertaking might not be safely assured. Often such development steps as key testing or evaluation procedures correspond to such important project milestones as Critical Design Reviews and Integration Testing completion. Surely it would be useful to the software engineer and project management to know that their development plans were sound at least to the extent that all needed inputs to these important subprocesses will be available to them. Data flow analysis seems capable of furnishing such assurances.

Closely related to the previous type of analysis is shortest path or critical path analysis. These types of analysis are already in use in the management of some (especially larger) projects. This type of analysis is aimed at detecting which development tasks and lines are most critical to assuring timely progress in a software project. The basis of these analyses is generally a network flow representation of the tasks of the project, annotated with estimates of the amounts of time and effort required to complete the various tasks. Our suggestion that the software development process be expressed as a rigorously defined algorithm will have the effect of making available the basis for the construction of a reliable network flow model, abstracted from the rigorous representation. Thus, critical path types of analysis would be placed upon a more rigorous and reliable footing, and would have concrete ties to actual ongoing development work. In fact, such flow networks would be good examples of other objects produced as part of the process of developing the software development process. Thus we see examples of other objects created by this process.

### 3.4.2. SOFTWARE METRICS IN EVALUATION OF SOFTWARE PROCESS-OBJECT DEVELOPMENT.

The preceding discussion also suggests a way in which to evaluate the performance of a software development process. The performance of a software development process might be measured in terms of the speed at which the process is executed (ie. how fast the development activity is progressing) and the size of the product being produced (ie. the size of the SP). It is important to observe that these issues are at the heart of the attempts that have been made for decades to attempt to provide reliable and accurate measures of software product size, and development progress. Our perspective now indicates that questions about the size of a software product, and the degree to which it has been completed now appear to be questions which should be addressed to the software process-object, rather than to the software product (SP), or worse yet,

to executable object contained within the SP.

If software development is aimed at the production of a software product which is a large and tightly interconnected structure of subobjects, then the size of the project must be a function of that object. If the degree of completion of the development process is to be measured, it seems only reasonable to approach this by measuring the extent to which the development process has proceeded through execution of the algorithm which describes it, and the extent to which the software product under development has been evaluated. Thus we are convinced that our perspective on software engineering provides important insights and directions for those who are interested in measurement and evaluation of software. We also see from this discussion that such measures are further examples of objects which should be created as part of the process of developing (and certainly maintaining) the software development process.

### 3.4.3. SOFTWARE MAINTENANCE PROCESSES FOR SUPPORTING REUSE.

Earlier we observed that software maintenance is an activity which is aimed at the alteration of either a particular software product, or either the structure or process according to which software product instances are developed. The difficult part of maintenance is not making these alterations, but rather making them while simultaneously assuring that as many of the previously developed software objects within existing software products can be reused. Thus it appears that software reuse considerations lie at the very heart of the difficulty in software maintenance.

Although not strictly essential, we believe that, in order for a software maintenance process to be acceptable it must be able to reuse, to a very extensive degree, software objects within products developed by processes which have been altered by maintenance activities, and are no longer in use. This, in turn, seems to be merely one manifestation of the more universal need to be able to effectively reuse in a software development (or maintenance) process software products (either complete or fragmentary) which have been created by a different development process.

In software product maintenance, one must alter the product while retaining as much of the product which has not been directly altered as possible. Here the primary difficulty is in determining the extent to which alterations have invalidated previous work. Clearly the materialization of the structure of the software product is an enormous help, as it can be used to indicate exactly which software objects must be related to which others and the exact nature of the required relations.

Thus one major functional requirement of a maintenance process is for it to

facilitate the detection of all software objects not affected by the maintenance changes and to recertify them for continued use in the altered software product. In the case of software objects which are affected by maintenance changes, the process must determine how these objects are affected by the alterations. The process should also determine which objects must be recreated, and the way in which the recreation should be carried out. In short, achieveing effective software product maintenance is dependent upon determining how to achieve effective reuse.

Similarly effective software process maintenance also depends upon effective reuse in the face of more significant alteration and disruption. In this case, the algorithmic process or the object structure of the software product is itself the object of change, and can no longer be considered to be a reliable constant, capable of guiding changes to underlying software products. The type structure itself may have undergone change, and may now describe a product which contains some new object and relation types and instances. In addition, the new type may no longer incorporate some object and relation types and instances which were incorporated into the previous one.

The job of the maintenance process is no different in this case. Here too, the process must provide effective support for the determination of which software objects, and which of their relations, have remained useful in the face of the changes, and which have been affected by the changes. In the case of objects and relations which have been affected, the process must provide effective support for determining the impact of the changes, must determine which objects and relations must be redeveloped, and the order and fashion in which such redevelopment are to take place. Object and relation types and instances which were present in the old SPS, but not present in the new SPS would not appear to be worth retaining. On the other hand, both the volatility of most software projects, and the proliferation of software projects suggest that software objects produced by a former development process have some future value, not anticipated at the time of development, and possibly unanticipated at the time of a decision about whether or not they should be discarded. Certainly the exceedingly high costs of software development should encourage the very careful consideration of possible future reuse of such software objects and processes.

Thus the larger software engineering issue to which our consideration of software maintenance leads is the issue of how the software products produced by one software development process can be reused as part of another software development process. In the classical conceptualization of software maintenance, the two software product structures in question are presumed to be quite similar, representing the (hopefully orderly and gradual) evolution of the development process. The software product itself is presumed to be incompletely

developed, raising the complicating issues of how to carry on with an algorithmic process on an object which was left only partially created (and perhaps in an internally inconsistent state) by a different algorithmic process.

In the emerging conceptualization of software reuse, what is needed is the ability to assimilate into a developing software product either part or all of a different product. Little attention has, to date, been focussed on the importance of the relationship between the process by which the candidate for reuse was developed and the process under which the developing software is being developed. It has been presumed, however, that the reused software product is to be made available to the reusing process in a "completed" form. Our current view of the nature of a software product is that it is a tightly interrelated structure of smaller software objects. Thus, the software product which is to be the reuser will have to assimilate the reused software into such a structure. Clearly, this can be done most effectively if the reused software is already built into such a structure. Thus, the issue of the process by which the reused software has been built now emerges as a critically important one. Small wonder that many who have attempted to reuse software have come away from their experiences disappointed and disillusioned.

This discussion indicates that effective reuse of a software product requires that both the product and the process by which it was created be delivered to the reusing party. The reuser, like the maintainer, is then in a position to compare the development process used, and the SPS which this process seeks to instantiate, and evaluate to the current development process and its SPS in order to determine how closely the algorithms, and object and relation types and instances match. In the case of a close match substantial reuse should be expected. In the case of a serious mismatch, significant benefits from attempts at reuse should not be expected.

Software managers who have attempted to reuse software have long ago understood that it is necessary to "plan for reuse" if this is to have a chance of success. Here we go further and note that such a reuse plan can and should be expressed procedurally in the form of an algorithm for the manipulation and evaluation of both contributed and existing software processes and products. Issues surrounding the construction of such reuse algorithms and the design of languages in which they can be effectively expressed would appear to be critically important research issues.

### 3.4.4. IMPLICATIONS FOR SOFTWARE MODULARITY.

Earlier we observed that the nature of a module is that it is a body of software objects which are relatively tightly interconnected among themselves, but relatively less tightly connected to other objects in the software product which contains them. One advantage of taking care to aggregate software objects into modules wherever possible is that the sparse connections of such modules to other software objects makes it correspondingly easier to combine them with those other software objects to comprise larger software products. Another advantage can be seen from our present discussion of reuse. We note that the relatively sparse connections between modules and the software objects of their containing software products also facilitate the process of reusing the modules. As the modules themselves are tightly interconnected lumps of software objects, the amount of software which is thereby reused is correspondingly larger as well.

In earlier work (eg. [Parnas 72]) the objects comprising a module were thought of as all being of type source code. From our perspective we now see that in a superior software product the source code objects are likely to be tightly connected to a variety of objects of other types. Thus we are led to conclude that a module must be a collection of software objects having diverse types. Further, this suggests that a collection of software objects may appear to be more or less modular depending upon the structure of the SPS which defines the software product within which it is embedded. Thus a collection of software objects created by one process may lack significant object types required by a different process. This lack may necessitate such considerable amounts of new software object and relation development that the software collection may not merit the designation "module." Conversely, a module may be furnished which contains a larger variety of object and relation types than the process which is receiving it either requires or understands. After these unfamiliar object and relation types and instances are ignored by the reusing process, it may be unclear to that process why the submitted module was considered to be a module at all.

One intriguing possibility that is raised by the suggestion that software products be created as elaborated, evaluated type instances is that the modularity of a subset of the objects comprising such a product might be measurable. This would presumably entail attaching measures to the various costs of elaborating the consistency relations connecting the various module objects both to each other and to objects outside the module.

## 3.5. SOFTWARE TOOLS AND ENVIRONMENTS AS INTERPRETERS FOR SOFTWARE ENGINEERING PROCESS PROGRAMS.

An environment should be thought of as a software system capable of effectively

assisting in the composition, compilation and execution of both software development/maintenance process programs, and the programs needed to develop and maintain such programs. Thus our work suggests that a software environment is a system of computer aids capable of providing effective support for the process of developing and maintaining the processes by which we develop and maintain software products. Further, we believe that this support ought to be supplied in the form of a single system offering the user as uniform and consistent a view of what it is doing as possible. We are convinced that an ideal vehicle for doing this would be a rigorously defined language, supported by compilation and execution support systems.

This perspective should help to clarify what the central issues in software environment design and implementation ought to be. If the environment is to appear to the user to be a language interpretation system, then some central issues are the efficiency of the implementation of the interpreter, the way in which the interpreter interacts with interpretive subprocesses, and the interface which these subprocesses and the interpreter itself present to the user. So far we believe we have established that an environment requires a very wide range of interpretive subprocesses, ranging from a subprocess to create specification to a subprocess for producing object code from source text. Thus we see that some operators must almost certainly be implemented manually (ie. by human processing) while others might well be better implemented by automated tools. In fact processes effected by automated tools are of particular concern to end users because the tools in question may be expected to be large and potentially quite slow. Thus, the implementation of the interpretive subprocesses is an important issue.

In our earlier work in connection with the Toolpack project [Osterweil 83, ClemmOst 86], we experimented with the implementation of larger tool functions by means of configurations of smaller tool fragments. This notion is quite consistent with the Unix (TM) "small is beautiful" [Kernighan 81] strategy. We discovered that by synthesizing large tool functions out of smaller tool fragment capabilities and saving the intermediate products of these tool fragments, we were often able to exploit these cached intermediate products in responding far more efficiently to future requests. Thus our new perspective indicates that this strategy of building larger tools out of smaller ones is an optimization strategy and it suggests that a key research area is the identification of the most strategically important object types and operators on those types.

The algorithmic specification of when operators are to be invoked is also important in managing both human operations and in managing computer operations. In the case of human operations, the algorithmic content of the development process is needed to specify the order in which human software developers are to go about their work. In the case of computer implemented operators

algorithmic direction is also quite useful in directing, and also controlling, the extent to which the software tools which implement the computer operators are to carry out their work.

For example, it is clear that a compiler should transform a source code object into the corresponding object code object, but it is less clear when this should happen. The obvious strategy seems to be to wait until the source code object is completely created. On the other hand, advocates of such tools as checkout and incremental (eg. [Gandalf 84], [DemRepTei 81]) compilers might well suggest that a compilation process or at least a parsing operation should be carried out automatically after each source line is created, or after some fixed number of such lines is created.

Pursuing this example further, consider the importance of exercising control over when consistency of object code objects should be determined by the relocatable loader. In the case of a very large body of source text, we often do not wish to study this sort of consistency by reloading the entire program every time a new source text object is compiled into an object code object. Control over the intervals and fashion in which this tool is to be applied ought to be specified by the software engineer and automatically carried out by the appropriate tools under the guidance and control of the development environment.

Our discussion has indicated that it is desirable for an environment to automatically invoke tools to create dependency and consistency relations, but that the environment ought not to decide how and when this should be done. This control must reside with the software engineer. The environment must then be a vehicle for assisting the software engineer in simply and precisely specifying this control and must then see that the software engineer's specifications are executed correctly. Our preference is for this specification of control of the application of software tools to be done through the mechanism of algorithmic programming in a software engineering language.

It is interesting to observe the extent to which some current software development environments match these design criteria. Clearly the Smalltalk (TM) environment [Goldberg 84] strongly encourages users to think of their work as being the process of creating objects and defining operations on types of objects. Similarly it seems that Interlisp encourages a user view of creating software by developing new objects and weaving them into a structure of existing objects [TeitMas 81]. Balzer's system [BalzerGN 84] and the Odin system also encourage this view. The last two seem to come the closest to widening their scope of object management sufficiently far to support development of large software. In the case of Smalltalk, the objects manipulated tend to be small grained, although this may not be inherent in the Smalltalk design and

approach. Interlisp does not supply mechanisms for typing objects, and thus seems to lack the capability for effectively controlling the application of tools. Recent work has, however, resulted in augmentation of Lisp to include typing mechanisms. Application of Interlisp or its architectural approach to such an enhanced dialect of Lisp might show promise as a vehicle for process encoding.

The strategies followed by these environments in controlling the application of tools seems to lack the flexibility and potential for user control which we feel is essential. In the case of Balzer's system and Smalltalk, the maintenance of consistency within the object store is carried out automatically whenever an object is altered. This was also the strategy pursued in the Joseph environment [Riddle 83] and was also a key design objective in Feldman's pioneering Make system [Feldman 79]. Thus automated tools are always invoked when it seems that they can be used to ripple out the effects of changes and thereby restore consistency. In Odin, the opposite strategy is used. Odin follows a lazy, or demand, evaluation strategy in which consistency and dependency relations are not maintained until and unless a user requests access to an object which participates in such a relation. At that time, all consistency and dependency relations involving that object, either directly or indirectly, are recomputed or reverified. Each of these strategies has advantages and disadvantages. Neither should be adopted to the exclusion of the other. Instead, the software engineer must be given the power to algorithmically specify how the two should be blended. A software engineering language in which the software development process is algorithmically described provides the basis for this control.

## 4. RELATION OF THIS RESEARCH TO OTHER KEY RESEARCH ACTIVITIES.

It is important to explain the relation of our research to activities in related areas. Earlier we observed that our attempts to understand key software engineering processes fits nicely with recent attention which is being devoted to studying the Software Process [SPW1 84, SPW2 85]. What distinguishes our suggested approach from most other work in this area is that we are attempting to use programming and program language formalisms and techniques to elucidate software processes in far more detail than seems possible without such formalisms. We believe that there is much promise in this work because it has already led to deeper insights into these processes and unconvered new relations between them. It also provides a mechanism for exploring and exploiting the exact nature of these relations.

Another distinguishing feature of our work is that it is aimed at developing the understanding needed to provide a firm basis for tool integration and environment architectures. This need to discover sound tool integration architecture

strategies was the original impetus for our search for process description formalisms. It has led to a somewhat new research orientation towards process and language studies themselves, but our orientation towards tools remains. Our ability to successfully pursue research in both directions more or less simultaneously is perhaps the most unique feature of this research.

### 4.1. KNOWLEDGE BASED SOFTWARE ENVIRONMENTS.

No attempt to characterize software environment research can be considered satisfactory without discussing its relationship to work in the area of Knowledge Based Software Environments (KBSE's) and Software Assistants (eg. [BalzerGN 84, SmithKW 85, Waters 85]). These exciting and innovative research projects have as their goal a new generation of software engineering aids in which knowledge is effectively captured, organized and then used in helping the software practitioner to do a better job. The philosophies and goals of these efforts are well summarized in [BalzerGrCh 83] and [TSE11 85].

From our perspective, the goal of KBSE research is to materially assist the various software engineering processes by automatically creating new information objects of importance to the processes. New information objects are automatically inferred from existing objects (possibly with human assistance) and that information is then applied--either with or without the knowledge and control of human parctitioners--to the creation or management of the ongoing development, maintenance or evaluation process. Thus we believe it is quite reasonable to characterize the goal of this research as the understanding of what is required in order to incorporate such automatically synthesized objects effectively into software processes. From our viewpoint, this research is aimed at the creation of daemons, capable of executing in parallel with other software engineering tasks, aimed at the synthesis of such new objects, and their effective assimilation into existing software product structures in such a way as to provide significant benefits to human practitioners.

This viewpoint suggests that KBSE research is quite compatible with our own, but would add an exciting new dimension to environments which we believe we can create now, with more classical tool supports. Current KBSE research is actively experimenting with non-algorithmic language paradigms for the expression of the subprocesses needed to assimilate and create new knowledge. We see no essential conflict between the use of one language paradigm for expressing the actions of (some) daemons and the use of another language paradigm for expressing other activities in a software process. What is important is that all activities operate upon the same information repository. KBSE research is currently aimed quite directly at coming to understand what the nature and structure of that repository must be. Clearly any advances in that research will

be of the utmost importance to us as well. We see no reason to doubt, however, that any structures which might be indicated by KBSE research will be eminently describable and constructible with algorithmic languages of the type we are considering.

Conversely, we believe that our own research should benefit KBSE researchers as well. Their work is aimed at effective creation of the most effective new information objects. Clearly the notion of "most effective" must be at least partly relative to the process which is to be aided by this information. Thus, we believe that KBSE research can itself be materially assisted by the deeper understandings of software processes and products to which we believe our own research can lead.

Another apparent difference between our work and KBSE research is that the knowledge objects which are currently the focus of KBSE research are much smaller than the large-grained software objects we have been describing. KBSE research is aimed at inferring knowledge of all types from objects of all types. Currently its algorithms tend to entail large amounts of processing. Thus at present it addresses the creation of relative modest sized information objects from objects of similar sizes. It is a clear goal of KBSE research to synthesize objects of whatever sizes are needed from whatever sized objects are available. This is not presently a practical goal, and KBSE research is currently thoroughly enough challenged by restricting itself to smaller objects. Our own research must eventually address the algorithmic description and programming of small software objects when refinement of these algorithms leads to low enough levels. Currently it seems most worthwhile to proceed top-down and thus come to an understanding of the overall architecture of key software processes. As our research leads to more detailed algorithms it will be necessary to precisely and effectively describe objects whose sizes range considerably, and operators which are probably hierarchically structured. We expect that doing this in such a way as to still enable effective and efficient execution of the resulting software processes will be an important and difficult research challenge. Thus, it appears that the difference in granular sizes of objects currently addressed by KBSE researchers and our own research is currently a matter of history and convenience. In the near future both research efforts will have to come to grips with the serious problems posed by widely different object sizes.

In summary, we believe that our research is nicely complementary with KBSE research. We believe that successes in both areas will be mutually enriching and stimulating. We see concrete ways in which we can profit from successes with KBSE's and we also see ways in which KBSE researchers might materially benefit from the results we expect to produce.

## 5. FUTURE DIRECTIONS FOR THIS RESEARCH.

In this section we outline plans for pursuing the research consequences of the view of software engineering just advanced. We suggest that diverse areas of software engineering research can be significantly advanced by establishing a language in which software engineering processes can be effectively expressed, by expressing various key processes in that language, by creating compilation and runtime support systems for that language, and by then attempting to use these systems to provide effective automated support for the execution of these key software processes.

### 5.1. EXPERIMENTATION WITH SOFTWARE PROCESS ALGORITHMS.

If there is a central focus to this research it is to devise and study various algorithms for expressing software engineering processes. We have already begun to create algorithms for software development, software product maintenance, software process maintenance, software product evaluation and software process evaluation. As we have proceeded with the iterative refinement of these algorithms we have begun to learn much about the nature of these processes, their relations to each other and to other processes such as reuse. Thus we expect that continuation of this activity will lead to more important insights, and, eventually to sound process algorithms.

Our intention is to arrive at acceptable algorithms for describing development, maintenance, evaluation and testing, and reuse. Having devised these algorithms we expect to solidify our notions of superior language paradigms and constructs for expressing them, and to solidify our notions of tool and environment architecture by examining the runtime structures and procedures needed to effectively support execution of these algorithms.

### 5.2. DESIGN OF THE SOFTWARE ENGINEERING LANGUAGE ITSELF.

While exploratory development of process algorithms seems more central to the pursuit of this research, development of a language system in which such algorithms might be encoded also strikes us as important. As this area is somewhat better established, it is, moreover, easier to categorize and organize the way in which this research might be pursued.

In this section we attempt to indicate how an orderly attack on the problem of creating a software engineering language might be organized.

### 5.2.1. THE LANGUAGE PARADIGM.

Earlier sections of this paper have indicated that some of the features this language must have are:

1. The ability to declare objects to be of different data types which are built into the language and to define new data types.

2. The ability to aggregate data objects into arrays and structures

3. Strong support for the creation and management of modules

4. Support for concurrency and tasking, in addition to alternation, looping and proceduring

5. Capabilities for defining and maintaining consistency and derivation relations among data objects.

This list of needed capabilities suggests that the language we seek requires that powerful tasking and synchronization capabilities be incorporated comfortably with powerful capabilities for creating highly dynamic, intricately interconnected data aggregates. It seems clear that the definition of such a language is an important research task which appears to be essential to the emergence of software engineering as a true discipline.

The first task in defining this language will be to decide which linguistic paradigm is most suitable. Earlier in this paper we gave some simple example algorithmic representations for some very high level software processes. These examples clearly betrayed a bias towards the use of an algorithmic language. This bias is at least largely based upon our belief that software engineers will be drawn from the ranks of software practitioners, and are most likely to be trained in programming in algorithmic, sequential (possibly parallel) programming languages. Thus it is most attractive to suggest that, as software engineers, they program in a language with a similar, and therefore comfortable, philosophy and paradigm. Making such an important decision based solely on the grounds of tradition and convenience is imprudent, however. Thus we have attempted to seek deeper justification for this prediliction.

Our second basis for believing that a parallel, sequential algorithmic language is best suited for programming software engineering is that software development and maintenance are to be carried out by human practitioners, and humans seem to us to be psychologically most self assured in thinking about their plans

and actions in terms of discrete, sequential steps and activities. Thus, a programming language which will be use to describe, regulate and control their activities in building software should seem most natural and comfortable if it expresses their activities in similarly discrete and sequential steps.

Although our current predisposition is towards more traditional and comfortable algorithmic languages, we recognize the need to consider non-traditional languages as well. We have already indicated a concern that traditional languages may not be effective in expressing consistency relations, for example. We have, accordingly, pondered a variety of non-algorithmic language paradigms. One interesting paradigm, for example, is that of a process control language. As previously described, software development and maintenance might well be viewed as a real-time processes involving the synchronization of the activities of people and mechanized aids. Borrowing from the idiom of process control software might thus be highly effective. From this perspective, activities such as code and design creation would correspond to synthesis (input) processes, and consistency checking activities would correspond to analysis processes. The software development program would then be a software system which incorporated a variety of such synthesis and analysis processes as asynchronous tasks, which, nevertheless communicated broadly among themselves, and paused at programmed intervals and events to synchronize and evaluate progress and consistency. Thus, control process software should be examined carefully as a model which might be worth emulating. Simulation languages capable of supporting the programming of such processes (eg. Simula and Simscript) are also worth examining.

Other potentially useful paradigms include object-oriented languages, functional or applicative languages and database language approaches. The appeal of an object oriented language approach is that it would clearly support and encourage the view of a software product as a collection of objects of diverse types. It seems clear that the designer of a language for software engineering should borrow heavily from object oriented language mechanisms for defining object types and operations on such types. In fact this strategy has been adopted by some KBSE researchers. Their work centers on the creation of a large and intricate knowledge structure which captures and correctly interrelates all information pertaining to the software being developed or maintained, and using that knowledge effectively in support of these processes. This is important to us, because, as observed earlier, our SPS can very reasonably be viewed as a knowledge structure, although it seems that the SPS and SP's we envision would probably be structures of much larger objects than are envisioned as the constituents of KBSE's.

We are still not persuaded that we should adopt an object-oriented language paradigm for our work, however, because of our relatively greater emphasis on

the algorithms needed to develop and maintain the knowledge and information structures which we agree are central. We remain convinced that software engineers and practitioners do maintain a strongly algorithmic view of what they do, but that they have been thwarted in effectively exploiting it due to the lack of an adequate expressive device. Object oriented languages do not seem to us to sufficiently encourage the attention to algorithmic expression which seems urgently needed at this stage of exploration of the nature of software engineering processes.

The appeal of a functional language is that it could support and encourage the view that software processes (eg. development and maintenance) are essentially the evaluation of large functions which are computed by the evaluation of a complex substructure of smaller functions. This view is appealing in that the software product which is the focus of these software processes is a complex composite of smaller objects and interrelations. Thus, it seems quite useful to describe processes on this product as the processes of creating its subcomponents and evaluating needed interrelations.

There appear to be significant problems in adopting a functional programming language approach, however. One is the need to at least partially linearize the order of evaluation of functions and subfunctions. The problem here is that some of the functions are to be carried out by humans who have difficulty carrying out unbounded parallel activities, and because the other functions must be carried out on a bounded number of computing devices. Functional programming systems assume the responsibility for such linearization, and take this process out of the hands of the software engineer. This strikes us as being awkward, at least for the present, when efficient compilation of efficient object code for large and complex functional programs is very much a research topic. Further, we continue to believe that some aspects of at least some software development processes are more straightforwardly describable in terms of sequential algorithmic steps rather than composition and nesting of functions. Testing and consistency determination would seem to be in this category. Thus, perhaps it is most reasonable to design the software engineering language in such a way as to combine both procedural and functional programming capabilities in such a way as to exploit the strengths of each in supporting software engineering process description and control.

Finally, it seems certain that at least some of the notational and descriptive devices used in database languages are useful as means for describing the software product in a software engineering language. Thus we would expect to borrow at least some of the descriptive mechanisms of such languages. We are not as sanguine about the prospect of exploiting such languages as vehicles for expressing software processes. We are particularly skeptical about how well such languages and associated support systems would be able to support

software process maintenance. Earlier we observed that this sort of maintenance entails alteration of the software product structure (the database schema) while retaining and reassimilating most, if not all, of its contents. We believe that extensible algorithmic language compilation systems currently provide the most useful paradigms for how to approach this problem.

Whatever the language paradigms used or merged to form the basis for the software engineering language, there will have to be important further research in establishing an effective semantic base within that language for support of software activities. Thus another important aspect of this research will be the determination of the built-in primitive data types and operators which the language should provide, as well as the appropriate linguistic and conceptual treatment of the relations which bind software objects together to form software products.

### 5.2.2. A COMPILATION SYSTEM FOR THE SOFTWARE ENGINEERING LANGUAGE.

As noted earlier, one of the primary reasons for creating a software engineering language is to use it as the basis for coordinating the work of software tools and their integration into a cogent software environment. A prerequisite for such an environment to be effective is the implementation of a compiler for the language. This compiler will have to accept SPS type declarations and transform them into object stores which will then be able to organize and structure software products as they are created. The compiler will also have to accept development and maintenance process definitions and transform them into procedures which coordinate the work of human software workers with each other and with the activities of software tools which implement various automatically supported software operations.

Compilation issues can be divided broadly into three types--syntactic issues, semantic issues and code generation issues. The last two types of issues seem to be the most interesting.

#### 5.2.2.1. SOFTWARE ENGINEERING LANGUAGE SEMANTIC ISSUES.

It is difficult to guess at which issues will pose the most difficulty in carrying out semantic analysis of the software engineering language, especially in view of the fact that not even the language paradigm has been selected. On the other hand, it does seem that a powerful and extensible type structure is essential, and this indicates that the semantic phase will have to be capable of potentially sophisticated type checking. In addition, the need for supporting extensions and alterations to the type structure of the language, while facilitating the large-

scale retention of objects created under the previous type schema, indicate the need for a compilation system in which the type structure can be modified as an object and used to create a new semantic phase for the compiler. In addition, the compilation system must be capable of analyzing the differences between the old and new type structures to enable maximal reuse and retention of objects.

This last observation suggests that the compiler for the software engineering language must be considered to be part of the process by which process-objects are developed. Any single instance of a software engineering language compiler must be considered to be an operator in the process by which process-objects are developed. In addition, however, the part of the software engineering compilation system which is able to alter the semantics of this language must be considered to be an operator which is used by the process which maintains process-objects. Thus the semantics-alteration system we have described is a process-object maintenance operator. It is interesting to note that we are now coming to identify some of the objects and operators which populate the process-object development and maintenance product and attendant processes. As we do so we are interested to note that they do not seem too dissimilar to objects and operators which we wish to describe with the software engineering language itself. This reinforces our belief that the software engineering language is suitable for maintaining both itself and the processes which it is used to encode.

#### 5.2.2.2. SOFTWARE ENGINEERING LANGUAGE OPTIMIZATION ISSUES.

The task of the optimization phase of a software engineering language compilation systems is to emit sequences of instructions to carry out and synchronize either human operations or computer based activities in such a way as to effect the algorithmic processes described by the coder. Language semantic definitions should assure that there is no doubt about how language operations are to be interpreted in terms of manual processes and mechanized tools. Further, control flow and synchronization operations will also require semantic definition in order to enable emission of effective object code.

Generation of efficient code is a more interesting problem. Two efficiency issues suggest themselves--one is the efficient storage of software objects and the second is effective reuse of intermediate software objects. The problem of achieving efficient storage of software objects is an important one, which seems to fall more in the province of runtime support systems, and will be discussed shortly. The problem of achieving effective reuse of intermediate objects is a central issue in compilation and also affects the philosophy of tool implementation. Reuse of intermediate objects is only possible if the operations specified

by the user can be seen as being composed of lower level operations which create such intermediate objects. Thus, if all of the operations in the software engineering language are implemented as monoliths, there would seem to be correspondingly little opportunity for reuse of intermediate objects. On the other hand, if operations are generally implemented as concatenations or structures of lower level operators, which produce intermediate software objects, these then become ideal candidates for reuse in subsequent computations.

Thus, the desirability of optimizing the object code generated by the software engineering language compiler provides strong impetus for the implementation of functional tools as composites of smaller, lower level tools (called tool fragments in [Osterweil 83]). The strategy governing the way in which intermediate objects are selected for storage for potential reuse was a research issue in the context of [Osterweil 83] and [ClemmOst 86]. Language statements were processed essentially interactively and there were no alternatives to statistical approaches to guide the strategy for saving intermediate objects. In this proposal, with our suggestion that software development and maintenance processes be captured in compilable code, it becomes clear that optimization algorithms and strategies much like those used in classical languages can and should be applied.

### 5.2.3. RUNTIME SUPPORT FOR THE SOFTWARE ENGINEERING LANGUAGE PROCESSOR.

It is clear that the software engineering language will require powerful runtime support subsystems in order to be the basis for effective execution of software engineering processes. Two key areas of support immediately suggest themselves for early consideration-- object management and input/output.

The need for a powerful object manager has been amply indicated by earlier sections of this paper. One of the central concepts in the approach we are suggesting here is that software development be thought of in terms of the need for creating, organizing and managing software objects. Clearly it is imperative to have effective ways in which to store them. The problems in doing so are badly compounded by our contention that objects are tightly interconnected to each other by such types of relations as hierarchy, derivation, and consistency. Clearly such simple organizational strategies as tree structures are woefully inadequate. We believe that relational database approaches have serious drawbacks as well. Some of these have been indicated earlier, and center on the dynamism of the structure of the object store.

We view the Odin system [ClemmOst 86] as a prototype object management capability which incorporates a number of desirable features. From the

perspective of this paper, we now understand that Odin actually incorporates some features of a software engineering language subset, a semantic analyzer modification and maintenance system subset, and an object manager. It seems clear that whatever object managment system is incorporated into the proposed software engineering language, it will have to have strong ties to the semantic analysis phase of the language compiler.

Input/output capabilities for the language are also quite interesting to ponder. Here we are inclined to view all processes which are carried out by humans as being input processes, and therefore in need of language assistance. Such assistance would have to range from simple text I/O support, through interactive editor support, to support for interaction with graphical and pictorial images. It seems essential that all of these interactions be implemented and supported in terms of basic language I/O primitives to assure a reasonably uniform user view of the software processing capabilities offered. Thus, whether the user were creating source code, design elements, test data sets or functional specifications, there would be a strong sense of uniformity of interaction with the software engineering language's features.

Output would have to offer a similarly uniform feel. The purpose of output capabilities would be to enable the user to see objects and relations in the emerging software product. Thus, we expect that it will be important for the user to be able to view a variety of objects, perhaps from a variety of perspectives, and to interact with those objects. This suggests that the I/O package will have to incorporate such capabilities as windowing, and menus. The use of color might well also prove to be of value.

Finally, it should be noted that the software engineer is also likely to need to view the process-objects which are being created and to get some insight into the processes which have been constructed. This interaction is different from the interaction needed by software practitioners. It corresponds more to the needs of a debugger of a program than to the needs of a user of that program. Thus, it is expected that the runtime system will also have to incorporate tools and capabilities for enabling the software engineer to study the structure and contents of the software process-object itself, in addition to the structure and contents of its individual component objects and relations. Here too, we are struck by the fact that these needs do not differ signifcantly from the needs of the software practitioners. This again suggests that the software engineering language may be adequate for the development and maintenance of programs for the development and maintenance of software process-objects.

## 6. CONCLUSION.

In this paper we have suggested that a reasonable focus of software engineering

is the notion of a "process-object"--namely an object which has been created by a development process, and which is itself a process. It then follows that the essence of software engineering is the study of effective ways of developing process-objects and of maintaining their effectiveness in the face of the need to make a wide variety of changes. These changes might entail alteration of the products produced by the process-object or alteration of the process-object itself.

The main features of the insights and suggestions presented here revolve around the notion that process-objects must be defined in a precise, powerful and rigorous formalism, and that once this has been done, the key activities of development, evaluation and maintenance of both process-objects themselves, and their constituent parts alike, can and should be specified and implemented algorithmically.

The suggested focus on process-objects draws a much-needed sharp line between software *product* development, evaluation and maintenance and software *process* development, evaluation and maintenance. This serves to improve our understanding of both and to help us to better understand the connections between such issues as maintenance, evaluation, reuse, and modularity.

These understandings are vital to further substantial progress in software engineering. We believe that they will be materially advanced by further research in identifying key software engineering processes and rigorously defining them by algorithmic specifications.

Finally, all of this strongly suggests the importance of devising a software engineering language and compilation/interpretation system. This language would become a vehicle for the specification of process-objects. A compiler for that language would become a vehicle for the organization of tools for facilitating development and maintenance of both the specified process, and the process-object itself. A runtime support system for the language would enable the execution of development, evaluation and maintenance processes, and would also provide a much needed mechanism for providing substantive support for software measurement and management.

We are convinced that vigorous research directed towards the specification of a software engineering language, its subsequent compilation and runtime support, and the use of such tools in the careful definition of key software processes will surely be of enormous value in hastening the maturation of software engineering as a discipline.

## 7. REFERENCES.

[BalzerGrCh 83] R. Balzer, T. Cheatham, C. Green, "Software Technology in the 1990's Using a New Paradigm," IEEE Computer pp. 39-45 (Nov. 1983).

[Balzer GN 84] R.Balzer, N.Goldman, B.Neches, "Specification Based Computing Environments for Information Management," Proc. Int. Conf. on Data Engineering, Los Angeles, pp. 454-458 (April 1984).

[Basili 80] V.R. Basili, Tutorial on Models and Metrics for Software Management and Engineering, IEEE Computer Society, New York, 1980.

[Bersoff 84] E.H. Bersoff, "Elements of Software Configuration Management," IEEE Trans. on Software Eng. SE-10 June 1984.

[Boehm MU 75] B. Boehm, R. McClean, D. Urfrig, "Some Experiments with Automated Aids to the Design of Large Scale Reliable Software," IEEE Trans. on Software Eng., SE-1, pp. 125-133 (1975).

[Boehm 81] B.W. Boehm, "Software Engineering Economics," Prentice-Hall, 1981.

[BryChaSie 81] W.Bryan, C.Chadbourne, S.Siegel (eds.) "Tutorial: Software Configuration Management," IEEE Computer Society Press, 1981.

[Clemm 84] G.M. Clemm, "Odin--An Extensible Software Environment," Univ. of Colo. Dept. of Comp. Sci. Tech Rpt. #CU-CS-262-84, Boulder, CO (1984).

[ClemmOst 86] G.M. Clemm and L.J. Osterweil, "The ODIN Environment Integration Mechanism," Univ. of Colo. Dept. of Comp. Sci. Tech Rpt. #CU-CS-323-86, Boulder, CO (1986).

[DemRepTei 81] A.Demers, T.Reps, T.Teitelbaum, "Incremental Evaluation for Attribute Grammars with Application to Syntax

Directed Editors," Proc. 8th ACM POPL, Williamsburg, VA, pp.105-116, (Jan. 1981).

[Dijkstra 72] E.W. Dijkstra, "Structured Programming," in Structured Programming (O.-J. Dahl, ed) Academic Press, NY, 1972.

[Feldman 78] S.I. Feldman, "Make--A Program for Maintaining Computer Programs," Software--Practice & Experience 9 pp. 255-265 (April 1979).

[FosOst 76] L.D. Fosdick and L.J. Osterweil, "Data Flow Analysis in Software Reliability," ACM Computing Surveys, 8 pp. 305-330 (Sept. 1976).

[Gandalf 85] Special Issue of "Journal of Systems and Software," on Gandalf Project. Journal of Systems and Software, 5, #2 (May 1985).

[Gilb 85] Tom Gilb, "Evolutionary Delivery versus the Waterfall Model," Software Engineering Notes, 10 pp. 49-61, (July 1985).

[Goldberg 84] A. Goldberg, "Smalltalk-80: The Interactive Programming Environment," Addison-Wesley, Reading, Mass, 1984.

[Guttag HW 85] J.V. Guttag, J.J.Horning, J.M.Wing, "The Larch Family of Specification Languages," IEEE Software, 2 pp.24-36 (September 1985).

[Howden 85] W.E. Howden, "The Theory and Practice of Functional Testing," IEEE Software, 2 pp.6-17, (Sept. 1985).

[Huff 81] K.Huff, "A Database Model for Effective Configuration Management in the Programming Environment," Proc. Int. Conf. on Softw. Eng, San Diego, IEEE Computer Society, pp. 54-61, March 1981.

[JeffTPA 81] R.Jeffries, A.Turner, P.Polson, M.Atwood, "The Processes Involved in Designing Software," in Cognitive Skills and Their Acquisition," (Anderson, ed.) Lawrence Erlbaum, Hillsdale, NJ, 1981.

[KafuHenr 81] D.Kafura, S.Henry, "Software Quality Metrics Based on

Interconnectivity," Journal of Systems and Software, 2 pp.121-131, (1981).

[Kant 85] E. Kant, "Understanding and Automating Algorithm Design," IEEE Trans. on Software Eng. SE-11, pp. 1361-1374 (Nov. 1985).

[Kernighan 81] B.Kernighan, "The Unix Programming Environment," Computer (May 1981).

[LeblChas 84] D.B. Leblang and R.P.Chase,Jr., "Computer-Aided Software Engineering in a Distributed Workstation Environment," ACM Sigplan/Sigsoft Symp. on Practical Soft. Dev. Envs., Pittsburgh, April 1984.

[LiskZill 75] B.Liskov and S.Zilles, "Specification Techniques for Data Abstractions," IEEE Trans. on Software Eng. SE-1 pp. 7-18 (1975).

[McCabe 76] T.J. McCabe, "A Complexity Measure," IEEE Trans. on Software Eng. SE-2 pp. 308-320 (Dec. 1976).

[Osterweil 81] L. J. Osterweil, "Using Data Flow Tools in Software Engineering," in Program Flow Analysis: Theory and Application (Muchnick and Jones, eds.) Prentice-Hall Englewood Cliffs, N.J., 1981.

[Osterweil 82] L.J. Osterweil, "A Strategy for Integrating Program Program Testing and Analysis," in Program Testing, (Chandrasekaran and Radicchi, eds.) North Holland, pp. 187-229, (1982).

[Osterweil 83] L.J. Osterweil, "Toolpack--An Experimental Software Development Environment Research Project," IEEE Trans. on Software Eng., SE-9, pp. 673-685 (November 1983).

[Osterweil 86] L.J. Osterweil, "A Structure for Software Engineering," University of Colorado Department of Computer Science Technical Report," to appear February 1986.

[Parnas 72] D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," CACM 15 pp. 1053-1058 (1972).

[Reps 84] T.Reps, "Generating Language Based Environments," MIT Press, Cambridge, MA, 1984.

[Riddle 83] W.E. Riddle, "The Evolutionary Approach to Building the Joseph Software Development Environment," Proc. IEEE Softfair--Software Development Tools, Techniques and Alternatives," pp. 317-325, Alexandria, VA, July 1983.

[Riddle 85] W.E. Riddle, Transcript of Symposium on Comparative Software Design Techniques, Aspen Colorado, July 1985. Contact Rocky Mountain Institute of Software Engineering, 1670 Bear Mtn. Dr., Boulder CO 80303.

[Smith KW 85] D.Smith, G.Kotik and S.J. Westfold, "Research on Knowledge-Based Software Environments at Kestrel Institute" IEEE Trans. on Software Eng. SE-11 pp.1278-1295, Nov. 1985.

[SPW1 84] Proceedings of Software Process Workshop, Runnymede, England, February 1984.

[SPW2 85] Proceedings of Second Software Process Workshop, Coto de Caza, CA, March 1985.

[Subra 85] P.A. Subramanyam, "The Software Engineering of Expert Systems: Is Prolog Appropriate?", IEEE Trans. on Software Eng. SE-11 pp. 1391-1400 (Nov. 1985).

[TeitMas 81] W.Teitelman and L.Masinter, "The Interlisp Programming Environment," Computer, 14 pp. 25-33 (April 1981).

[Tichy 83] W. Tichy, "RCS--The Revision Control System"

[TSE11 85] IEEE Transactions on Software Engineering, Special Issue on Artificial Intelligence and Software Engineering, (J. Mostow, ed.) SE-11, (Nov. 1985).

[Waters 85] R.C. Waters, "The Programmer's Apprentice: A Session With KBEmacs," IEEE Trans. on Software Eng. SE-11, (Nov. 1985).

```
Procedure develop_software;
Declare software_product  structure of
-
-At the highest level, perhaps only
-these three major components of the
-final software product are visible
-
    Requirements_spec,
    Design_spec,
    Code;
-
-This first piece of code represents
-the process of creating a plausible
-initial set of requirements
-
create_requirements;
-
-carry out a process of reviewing the
-requirements and determining that they
-are "ok", what this means is to be defined
-at lower levels of this program
do_while ~(requirement_ok);
    fix_requirements;
    od;
-
-Create a first set of design specifications
-
create_design;
-
and check that they are "ok"
-
do_while ~(design_ok);
    fix_design;
    od;
-
-Now build the code
-
create_code;
-
-Now make sure the code is "ok"
-
do_while ~(code_ok);
    fix_code;
    od;
```

```
-At this point there is a software product
-consisting of requirements, design and code.
-The next piece of this program addresses
-the way in which it is to be maintained.
-
do_forever;
-maintenance is to be described in a susequent
-section of this proposal. Here we merely indicate
-that it can be viewed as a process which is carried out
-subsequent to the development process.
maintain (software_object);
end develop_software;
```

Figure 1. A very simple, top-level algorithmic description of the software development process as suggested by the classical "Waterfall Model" view.

```
declare reqirements_specification
        consists_of
          array_of testresults;
          array_of testcases;
      testcases
        consists_of
          (test_input, results_required);
-
      testresults
        consists_of
           array_of results_produced;
-
```

Figure 2: The declarations necessary to support a simple algorithmic testing process.

```
procedure code_ok (requirement_spec, code);
declare
-
-declarations inserted here are as shown in Figure 2.
-For this procedure, we assume that TEST is an instance
-of type testcases, and the components of its elements
-are INPUT and RQD
-
-begin by creating the executable object derived from the
-code object being tested
-
derive (code.executable, code);
code_ok := TRUE;
do_for_all elements, I, in TEST
      derive (RESULT[I], (code.executable, INPUT[I]));
      if ~consistent (RESULT[I], RQD[I])
            then code_ok := FALSE;
                 exit;
      od;
```

Figure 3: A high level algorithm describing the way in which a code testing procedure can be expressed.

```
procedure create_code(S);
For_all design_specs DSO in design_object, D
     do
       get DSO;
       build source_code_object (SO);
       while ~ok(S,SO)) do
          case of:
            code wrong: change_code (SO),
            design_spec wrong: change_design (D),
            ok_spec wrong: change ok_spec,
               esac;
       od;
end create_code;
```

Figure 4: A very high level algorithmic procedure for describing the way in which evolving code is incrementally evaluated in practice.

```
procedure ok(S,SO);
declare S structure_of
        source_code_objects, SO;
if ~complete_code (S)
    then
      static_check(SO);
      create_test_harness (S,SO);
      create_testcases (SO);
      ok := TRUE;
      do_for_all testcases, T
         if ~consistent (T,SO)
             then
               ok := FALSE;
               exit;
             endif;
          endo;
    else
      ok_code (SO);
    endif;
end ok;
```

Figure 5: A very high level algorithm representing what is required in order to test a portion of an incomplete body of code.

```
procedure do_maintenance;
input (change_request);
case of change_request is
    requirements_change : change_requirements;
    design_change       : change_design;
    code_change         : change_code;
     esac;
end do_maintenance;
```

Figure 6: A very high level procedure indicating the overall structure of a software product maintenance procedure.

```
while ~empty(maintenance_request_list) do
        accept_next maintenance_request;
        process_maintenance_request;
-this last procedure presumably incorporates an invocation
-of a procedure for checking the consistency of the resulting
-software object
-
        delete maintenance_request;
        endo;
```

Figure 7: A simple dispatching loop for servicing maintenance requests.

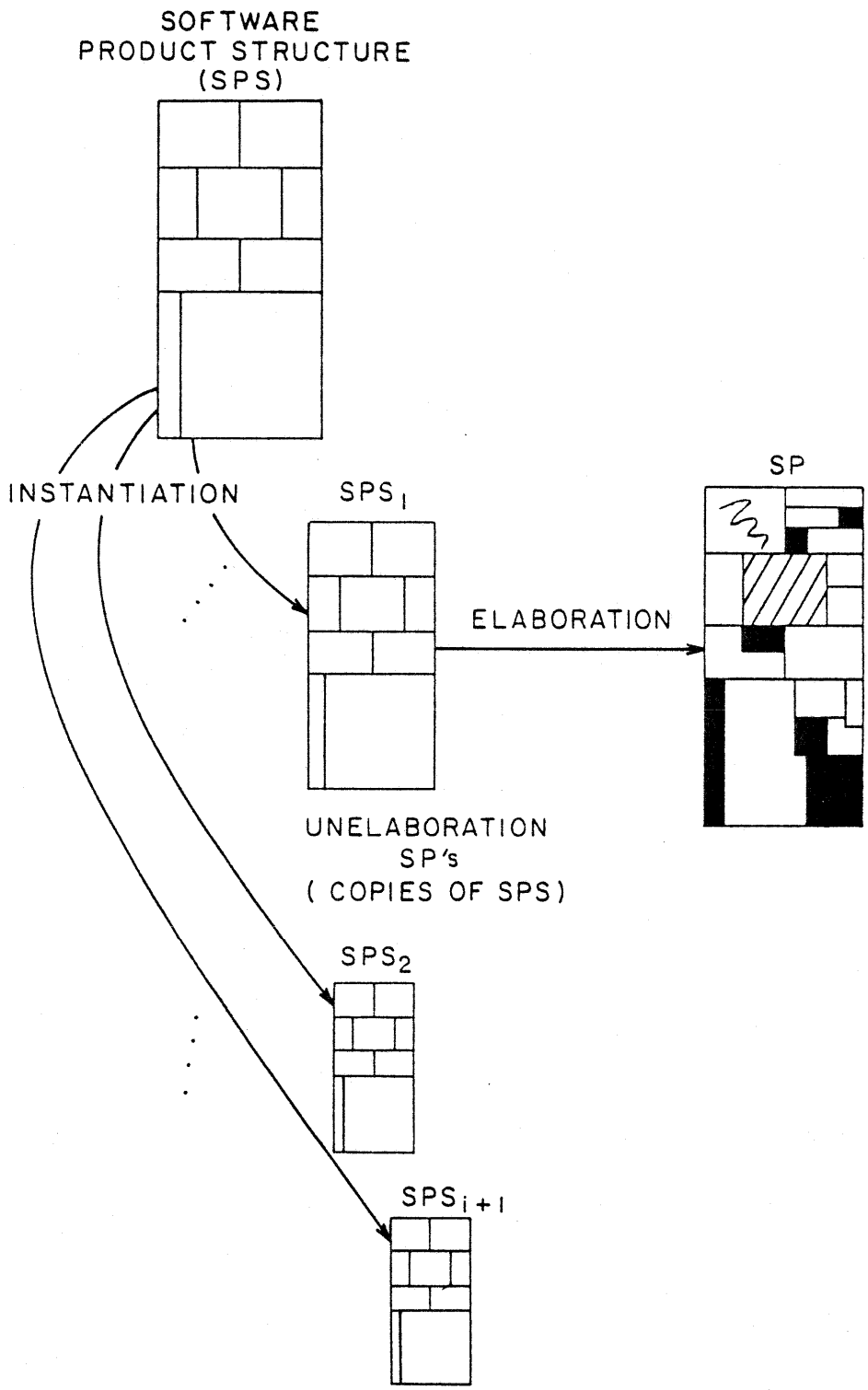



Figure 8: Software development, viewed as the process of instantiating an SPS and then elaborating the SPS instance into an SP.

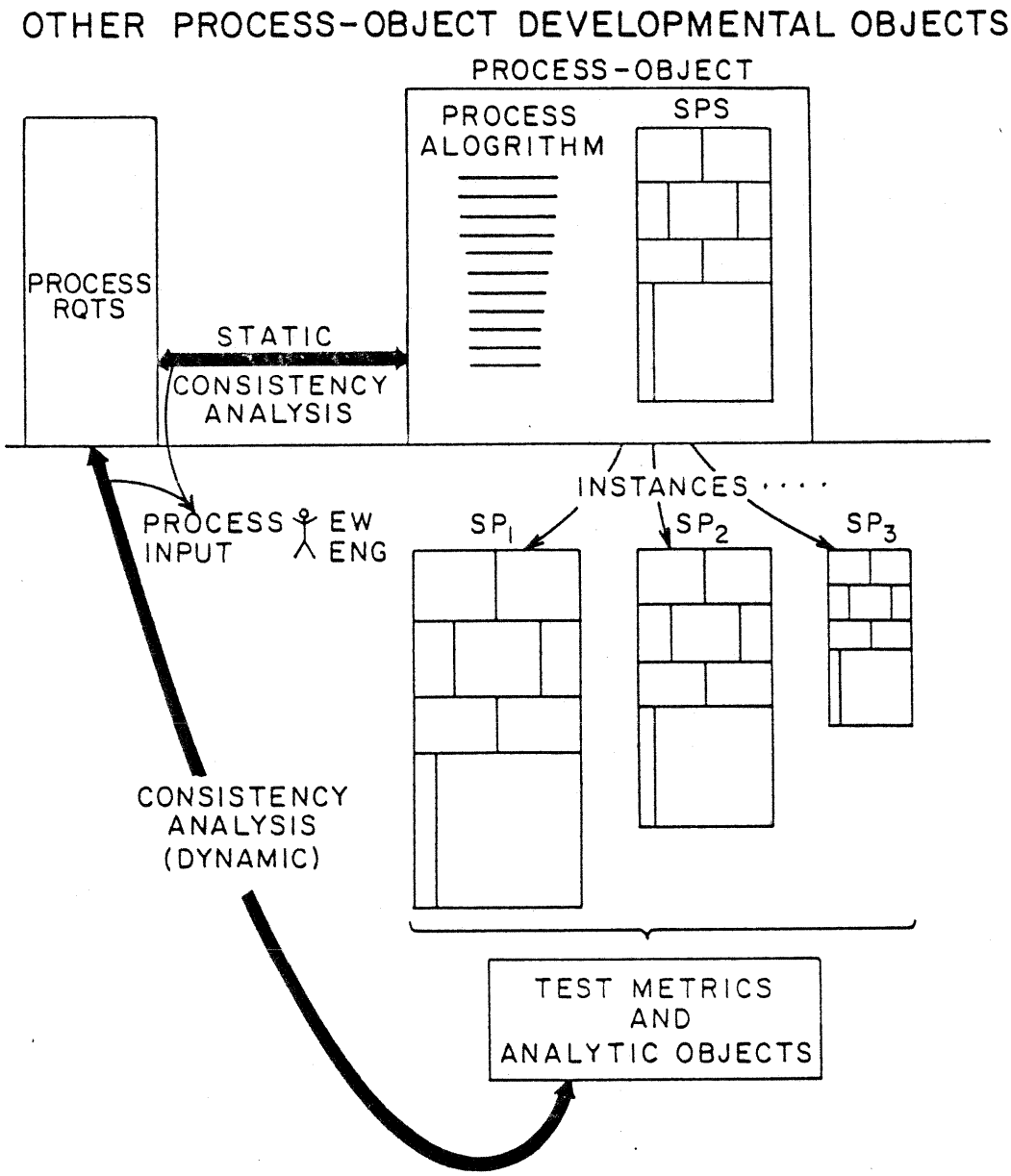



Figure 9: Process-Object maintenance: The Process-Object, created by an unillustrated development process, in response to a process-object object requirements specification, is evaluated both by static analysis of its structure and contents and by evaluating SP's which it produces (dynamic analysis of the process-object). Both forms of analysis are essentially consistency comparison of derived objects with original requirements object.