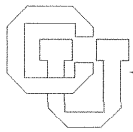
Software Maintenance as a Programmable Process *

Shehab A. Gamalel-Din
Leon J. Osterweil

CU-CS-390-88



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This research was supported by the Defense Advanced Research Projects Agency, through DARPA Order #610, Program Code 7E20, which was funded through grant #CCR-8705162 from the National Science Foundation, with additional funding through National Science Foundation grant #DCR-0403341.

Software Maintenance as a Programmable Process

Shehab A. Gamalel-Din & Leon J. Osterweil

CU-CS-390-88 March 1988

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

This research was supported by the Defense Advanced Research Projects Agency, through DARPA Order #610, Progam Code 7E20, which was funded through grant #CCR-8705162 from the National Science Foundation, with additional funding through National Science Foundation grant #DCR-0403341

Document Citation Numbers

University of Colorado, Boulder
Department of Computer Science
Technical Report Number CU-CS-390-88

Arcadia Document Number
CU-88-06

Software Maintenance as a Programmable Process

Shehab A. Gamalel-Din & Leon J. Osterweil

Boulder, Colorado 80309

The software maintenance process is a particularly complex part of the software life cycle. It can be viewed from a number of different perspectives and dimensions. The policies and philosophies of the maintenance organization and its management, the techniques available for carrying out maintenance, the types of changes attempted, the points in the development process at which maintenance is attempted, and the nature of the subject product are among the factors playing important roles in shaping and designing a maintenance process. No single fixed maintenance process seems able to meet all software maintenance needs emerging from the different perspectives and dimensions, and nobody has yet consolidated all of those views in a single framework. We believe that consolidating the maintenance activity around the notion of *"Process Programming"* provides such a common framework for all software maintenance processes. It provides the conceptual structure for creating processes and support environments in which users are free to alter both tools and process to achieve effective support for the full range of maintenance needs and approaches. *"Process environments"* -- environments which support process programming -- seem to us to meet the minimum requirements for an ideal environment. They focus on both describing and aiding the process itself in a customizable (programmable), user-tailorable, dynamically adaptable, and incrementally implementable fashion.

However, process programs are long term processes which encompass learning and exploration activities. Iterative and continuous improvements to processes are to be expected, even during the course of execution of the process. The continuous evolution of process programs during their execution adds new dimensions and perspectives to the maintenance process. Considering this has caused us to suggest on improved model of maintenance and a new software life cycle. *Process maintenance* and *dynamic maintenance* as well as the related topics of *process execution history maintenance* and *product_related process maintenance* are all newly introduced notions arising as a direct consequence of the characteristics of process programs. The new notions all combine to greatly complicate the activities of maintaining process programs themselves. This raises the need for maintenance environments and tools to support these new complicated activities. Thus, here we propose a sketch for a maintenance environment and tools to overcome the complications which these new notions introduce.

March 25, 1988

Software Maintenance as a Programmable Process

Shehab A. Gamalel-Din & Leon J. Osterweil

Boulder, Colorado 80309

1. Introduction

Maintenance is the process of designing and integrating consistent changes to the objects and/or relations of an existing and delivered software configuration [Schneidewind 87]. The problem of doing this effectively is very complex and has a number of dimensions which cause individual maintenance tasks to vary very widely in nature. The nature of requests for changes may vary radically. The resources available for making requested changes may also vary widely. Maintainers may have access to widely differing tools and environments to support their work. The size of the changes to be made may vary widely. The software product to be maintained may vary widely in size, makeup and complexity. Finally, the methodologies used to develop the product now under maintenance may have varied considerably. It is possible to consider maintenance from each of these different narrow perspectives. Doing so might suggest a different maintenance process for each perspective and hence, different environments for supporting maintenance might be suggested for each of these different perspectives.

This paper examines some of these perspectives and shows what characterizes the maintenance process for each of them. It then proposes a common framework -- process programming and process environments -- for characterizing all of them. Such a framework, in turn, leads to new understandings of software maintenance, and suggests changes to our classical understanding of this activity. For example, we have been accustomed to thinking of maintenance as being a process which is carried out on a purely static product. In this paper we widen our concept of maintenance, suggesting the additional need to consider the process as an essential concept in building maintenance environments. This improves our model of maintenance and adds new dimensions and perspectives. Two new essential perspectives are due to the need for maintaining the process as well as the products it produces, and the need to be able to alter software products while they are still under development.

We refer to the process of altering software under development as *product-related maintenance*. We also suggest that, while most maintenance research has focused on making alterations to software products it is necessary to also consider the need for making alterations to the processes by which such products are created. We refer to this as *process-related maintenance*. We note the need to alter both the static process description and the process in its running state. We refer to the alteration of the static process description as *static process maintenance*, and to the alteration of the executing process as *dynamic process maintenance*. Consideration of the nature and needs of dynamic maintenance and process maintenance as well as classical product maintenance leads to a new appreciation of maintenance as a very general and powerful process and to increased respect for the problem of creating the tools needed to support it.

Section 2 develops some understandings of the different maintenance perspectives, explaining their characteristics and differences. It addresses some of the problems common to all maintenance processes and maintainers. It discusses some earlier work, indicating the perspectives from which it has been approached, and it highlights some problems which are still not satisfactorally addressed. This leads us to suggest a common maintenance framework based on the notion of process programming [Osterweil 87]. This common framework is elaborated upon in section 3. The introduction of the new framework introduces, in turn, a new set of maintenance related problems which are also highlighted in section 3. Section 3 also discusses the differences between product and process maintenance and introduces the notion of dynamic maintenance. It then introduces a new paradigm for the software lifecycle which combines the development and maintenance processes under a unified process. A powerful maintenance environment which supports these new notions and perspectives is proposed in section 4. Examples and algorithms are given in appendixes.

2. Classical Perspectives and Background.

Software maintenance varies greatly in practice and perspectives within the software engineering community. Policies and philosophies vary, organizational and management structures vary, tools and techniques vary, even the activities referred to by the term "maintenance" vary. In this section we summarize some different classical views of maintenance, highlighting the diversity of approaches to maintenance, and the variety of research projects which have attacked it. We then present our own view of maintenance, suggesting that it must be viewed in a very broad perspective, and must be supported by a very large and flexible collection of tools. We focus on the central importance of orderly processes for carrying out maintenance, but observe that no single fixed process should ever be expected to be able to meet all the needs and perceptions encompassed by the term "maintenance." In section 3 we then present details of a framework and flexible tools which might be used to support the full range of currently understood maintenance needs, as well as some other maintenance needs which have not yet been clearly identified.

2.1. Technical Perspectives.

Many maintenance research projects have focused entirely on the technical aspects of maintenance, addressing only the process of performing changes. These efforts have supported everything from understanding the need for change, to modifying the software product, and to revalidating the software product [McClure 81, Martin 83].

There have been a number of attempts to understand the problems encountered during maintenance. For example, there have been empirical and statistical studies [Lientz 78, Peercy 81, Dean 83] of the problems and flaws of actual maintenance processes. Exhaustive lists of the problems encountered during maintenance have also been compiled [Lientz 80, Pressman 82, Kishimoto 83, Lientz 83, Martin 83]. Some examples of these problems are: 1) the difficulty in understanding "alien" code, 2) the need to assess the syntactic and semantic impacts of the ripple and side effects of modifications, 3) the identification of the structure of the different components of a software product, 4) the revalidation of changes, and 5) the estimation of resources required to carry out such maintenance steps as recompilation and relinking.

Most of these problems have been attacked both directly and indirectly. Direct attacks have concentrated on useful tools such as preprocessors, prettyprinters [Parikh 82], code restructurers [Balbine 82, Bush 85], control and data flow analyzers [Fosdick 76], and ripple effect analyzers [Shneiderman 86, Stankovic 85, Agusa 85]. This work has addressed important maintenance problems directly, but has also provided indirect help by leading to useful insights and knowledge into the software under maintenance.

Other indirect attacks have focused mainly on improving quality and maintainability [Martin 83, Schneidewind 87] of software products. For example, [Warnier 81, Parikh 86, Higgins 86] advocate structured code development. Chikofsky [Chikofsky 85] has explored increasing maintainability through design specifications written in specific formalisms such as PSL/PSA. Others, (eg. [Parnas 83, Gilb 83, Martin 83]) have attempted to improve the entire development process. Still others focus on the use of metrics to guide and improve the maintenance process [Herndon 83].

The activities just described take two approaches to improving maintenance. The second approach assumes that difficulties will be sharply reduced if software engineering and structured development are used to produce software. This approach, although arguably producing highly maintainable software, is unsatisfactory by itself, because even software produced in this way will require some modification at some time. This approach also provides no help in dealing with the large quantity of existing software which has not been systematically produced.

The first approach is more pragmatic, entailing the development of specific tools to solve specific problems. This is a limited approach, which does not provide a general solution to the maintenance problem. Perhaps its greatest benefit, however, is that each attack on one problem clarifies the need for other, larger efforts. For example, tools supporting program understanding are useful, but after using them one discovers that the understandings they convey are limited. None of them is able to assess the semantics of, say, a variable involved in a delocalized plan [Letovsky 86], or is able to describe dependencies among different program entities (e.g. types, variables, and parameters). Ripple and side effect tools work only locally. They do not consider the dependencies among the program entities inside a module or a compilation unit. Instead, they assume the existence of other tools for relating larger grained objects. These

realizations spawn work on larger more effective tools, but the realizations arise out of the development and application of more modest tools. In some sense the growing family of such limited tools can be seen as prototypes which are useful in helping us to see the underlying requirements for the maintenance toolset which is what we must eventually develop.

Unfortunately, these tools are rarely designed in a cooperative way so that they could be easily integrated for supporting a single maintenance process. For example, most incremental development environments contain structure editors which automatically check for the correctness of a change only from the compilation viewpoint. They neither support any other maintenance operations nor provide change information that other analyzers could use to update their knowledge bases. Thus we believe the greatest weakness of this second approach is that it is hard to merge and integrate the many tools which have demonstrated limited applicability.

2.2. Management Perspectives.

While computer scientists think of maintenance as being a technical process aimed at effecting careful alterations to software objects, managers understand that it also entails careful procedures to assure that such changes are not disruptive. Management and technical activities must be well integrated, even though management is generally executed by humans while technical activities are carried out by tools.

Maintenance management activities can be divided into two classes -- product-related and process-related. Product-related management is far better supported by computer aids, such as version and revision control systems (e.g. SCCS [Rochkind 75] and RCS [Tichy 85]), change coordination systems (e.g. Infuse [Kaiser 87, Perry 87]), reuse support systems (e.g. Draco [Neighbors 84, Arango 85]) and configuration control systems (e.g. Make [Feldman 79], Odin [Clemn 84], NuMil [Narayanaswamy 87])

Process-related management activities [McClure 81, Pressman 82, Martin 83, Parikh 86] include personnel management, resource management, subprocess scheduling, walk-throughs, quality audits, and planning, and are generally done manually, although some machine aids have been developed. For example, MONSTR [Cashman 83] provides a programmable protocol-driven personnel communication control tool which managers can use to monitor maintenance activities and status. MONSTR is particularly interesting to us because it supports a wide variety of different maintenance processes in different ways, by enabling the tailoring of communication protocols, specifiable within its modeling framework. We believe that flexible maintenance management systems, as exemplified by MONSTR, are the most promising ways to provide computerized support for the wide range of processes in use and projected. Further, we believe that such systems must also be integrated with tailorable product-related maintenance management systems which extend the capabilities described above and they must also be carefully integrated with the growing collection of tools which support the more technical aspects of maintenance.

2.3. Maintenance as a Process.

We believe that the task of effectively integrating such a large and diverse collection of tools must be focused by considering such tools to be mechanisms for supporting the conduct of maintenance processes. We believe that maintenance workers do and must think of their jobs as being disciplined organized activities which provide the context into which tools and aids must fit. Thus it seems particularly important for us to now summarize the sorts of maintenance processes which currently exist. We first categorize software knowledge acquisition processes, and then categorize principle processes for carrying out maintenance changes.

2.3.1. Knowledge Acquisition Processes.

An important distinction has been drawn between the process by which well structured software products are maintained and the process by which all other software is maintained. Pressman [Pressman 82] has defined Structured Maintenance to be the process of maintaining software which has been developed using systematic software engineering and/or other structured development techniques. This process assumes the existence of a wide variety of documentation objects which can be used to guide and facilitate maintenance chores. Thus structured maintenance describes any maintenance process which begins with assumption of the existence of a significant collection of objects produced as part of a systematic development process.

On the other hand, unstructured maintenance must build all software objects, and the understanding which they convey, from scratch using source code only [Fay 85]. These maintenance processes are obviously much harder to carry out, as every step entails the production of new information, much of which is available in the former method.

Unstructured maintenance processes should incorporate activities aimed at capturing the knowledge and software objects that are generated, but rarely do so. On the other hand it is important to note that even structured maintenance activities often suffer from this failing as well. While maintenance is facilitated by the availability of software objects such as design information which had been created during development, it is generally the complex relations among these objects which is of most value. This relational information is rarely available, even to a structured maintenance activity, and needs to be recreated by maintainers. Once it is recreated, it is rarely captured in explicit form.

2.3.2. Principal Software Alteration Processes.

There are a wide variety of processes which are used to carry out alterations to software. We have named these processes: backbone, spare parts, design recovery, copy-and-adapt, black-box, and reuse. Each process has its own requirements for support tools and levels of information and understanding. Different adaptations and combinations of these basic processes are generally used in particular maintenance situations. This suggests that no fixed maintenance procedure is likely to ever be universally satisfactory, and that a flexible framework in which special purpose maintenance processes can be tailored or adapted is desirable.

In *backbone maintenance* modifications are done by addition, deletion, and replacement on the body of the program but no alteration is made to its structure. This method requires detailed understanding of the code to be changed and a firm understanding of the side and ripple effects of the changes. Clearly this is most easily done as part of a structured maintenance process.

Spare parts maintenance [Gilb 82] entails replacing an entire code module or unit by an equivalent unit which has the same external specification and interfaces but which probably differs in internal design. This method requires an understanding of the functionality and interfaces of the overall module, but does not require an understanding of the internal structures and implementations.

In *black-box* maintenance new software is built on top of old software without trying to modify it. Using routine libraries is a good example. All this technique requires is an understanding of the old software interfaces--its input and output--and their relations to the new goal. A mathematical relation between the functionalities of the two is assumed, and the new software is a realization of that relation.

Design recovery [Arango 85] assumes that the software was produced by careful and orderly iterative refinement of a firm specification. It requires understanding all of the details of the refinement process and how the different revisions and versions of the software which may have evolved are related to each other. It might also require a knowledge base incorporating knowledge of the software product's subject domain and a tool for browsing and resolving this knowledge.

The fifth method -- *copy-and-adapt* -- is perhaps the best known. It entails modifying existing code to meet new or changed needs. When a maintainer discovers the need for a procedure which has a lot of similarities to another existing one, that procedure is copied and then changed in such a way as to be adapted to the new goal. This technique requires a thorough understanding of the existing procedure and the use it makes of all non-local variables, in order to avoid dangerous side effects [Letovsky 86].

Reusability techniques [Diaz 87] could be used in maintenance if they were used in the development process, and if they are supported by the appropriate tools and languages. We believe that development processes which are based on the notion of reuse rest upon the same basic needs and activities as those which must support effective maintenance. Thus development exploiting reuse might well be called development-by-maintenance.

The last three techniques discussed could be considered special cases of reuse. However, we prefer to use the term "reuse" for a different class of techniques in which the reusable components are conditionally combinable by a set of implicit or explicit operators. Object-oriented languages (e.g. Smalltalk-80 [Goldberg 83], C++ [Stroustrup 86]) are good examples of languages supporting such techniques. They provide great flexibility in combining reusable components produced in the same language. New software

inherits facilities from specified existing components, and may augment or replace any subset of inherited data structures or operations. The maintenance activity here requires understanding of the interface, the implementations of the reusable components, and combination methods. Inherited functionality and data structures can be modified by new components. The resulting components are usually added to the library of reusable components to increase the possibility of reusability still further. This technology requires large amounts of documentation, knowledge, and tools for browsing in order to be effective.

Each of the above techniques imposes different requirements on the product to be maintained, the amount of knowledge needed, and the support tools required. Hence, a different process must be devised for each technique, each requiring a different support environment. For example, a process using the design recovery technique must work on specifications which are assumed to be available in a hierarchical form representing the versioning and deviations from one version to another. While the reuse approach assumes the same hierarchical representation but requires no knowledge about different versions. The process in each case is quite different and hence so would be the environment supporting it.

2.3.3. Different Requirements for Maintenance Processes.

Maintenance processes have also been categorized according to the different ways in which maintenance needs arise [Swanson 76]. Each type of need has its own influence on the process and techniques employed and has its own requirements for support tools.

Corrective maintenance, for instance, usually entails making changes to code and perhaps design. It rarely affects specifications and test plans. It usually follows a debugging process, and modifications may be done on the backbone of the original program.

preventive maintenance is the process of making changes to assure that software will not behave incorrectly in the future. Often spare parts maintenance is used to do this, with existing unsatisfactory or suspect pieces of code being replaced by externally manufactured spares with minimum (if any) effect on the design of the larger software product.

Adaptive maintenance, on the other hand, is done to update the system so as to meet new requirements due to environment or processing changes. System functionality may not be changed at all. Often adaptive maintenance entails changes to nearly all components of the software product configuration. Usually it can be done by either spare parts or backbone techniques.

Perfective maintenance is carried out to add new features to an existing system or to improve existing features. This, too, often requires making additions or replacements (in the case of performance improvement) to every component in the software product and is usually done by design recovery or copy-and-adapt techniques which are facilitated by use of powerful global analysis tools.

2.4. Summary

Because of the bewildering variety of perspectives, approaches and research projects, no single consolidated view of maintenance has yet emerged. Because perspectives and practices vary so widely, there is a temptation to think that such a single encompassing view simply does not exist. We believe that there does not exist a single process that precisely meet all maintenance needs, and there does not exist a single standard toolset that precisely supports all maintenance processes. On the other hand, past experience in addressing specific individual problems has had the beneficial effect of elucidating the important maintenance issues, and does now seem to suggest a general framework for the solution of the maintenance problem as a whole. As shall be seen, this framework enables the flexible modeling of the widest range of maintenance processes, and facilitates the creation of effective tool supports for them in the form of powerful environments.

3. A Common Maintenance Framework and Support Environment

We believe that it is possible to build on the work and knowledge that has been gained in past software maintenance research efforts to devise a common framework in which maintenance can be understood and a common collection of tools which effectively support the spectrum of maintenance processes. The key to being able to do exploit the idea of integrating tools effectively into environments which support explicit processes. We note that tools provide automated support for specific tasks, but they generally lack uniformity, completeness, and compatibility. Tools are most effective when they are coordinated with each other in a larger systematic framework such as an environment. All environments informally define the processes that they support. Dowson explains that [Dowson 87] "most existing environments do have some implicit model of the process they support. In part, this is because any collection of tools will impose some restrictions on how they may be deployed; and in part because many environment designers build in their prejudice for some approach..." We agree with Dowson, however we suggest that what is needed is a flexible environment having an explicit representation for the process it supports, and in which users are free to alter tools and the process itself in order to achieve effective support for the maintenance needs that they have and the maintenance approaches that they fashion. These observations form the philosophical basis for the proposals that we will now advance.

3.1 Key Maintenance Environment Characteristics.

Before proposing a common framework for a comprehensive maintenance environment, we first present some characteristics which we regard as critical. In the next sections we will show how all of these characterizations are met in process programming environments [Arcadia 87, Osterweil 87].

- *Explicit process representation.* It seems ironic to us that the tools and environments discussed above focus on effective support of particular maintenance processes, without enabling users to have good visibility into those processes. Stenning in [Stenning 87] assesses the role of an environment saying "The overall objective is to achieve some combination of higher quality, lower cost, and greater predictability. This can only be done by *focusing on the process itself.* Improvements could only be achieved by introducing effective process into effective use." That is, an effective environment must be able to support an effective process. However, a firm understanding of the process and its characteristics is urgently required to assess the properties of an ideal maintenance environment. We define a *process environment* as "an environment which contains an explicit model describing the process that it supports".
- *Open for new tool augmentation.* Section 2 shows that some maintenance problems are addressed and partly solved by means of tools while others are not. This argues for the need for more tools. Hence, the ideal environment architecture must be *open* to accommodate those new tools.
- *Customizability* and *User-tailorability.* As discussed in section 2, maintenance processes vary tremendously depending on maintenance objectives, the skills of the personnel involved, the nature of the product, the availability of automatic support tools, and the specific maintenance project. So, any general purpose maintenance process must be *customizable (programmable)* to meet these diverse requirements. Hence the environment supporting the process must also be customizable. We believe that designers of maintenance processes should be given the ability to specify and customize their processes to whatever level of detail they desire. This implies that some processes will be very detailed, while others may be left largely unspecified. It is important that the role of every part of the process must be well defined when it is customized, even though some details of how to carry out various specific subprocesses may be omitted and left unspecified.[1]
- *Dynamic adaptability.* Maintainers are usually content to follow a given process until they feel it is inconvenient, inefficient, dangerous, or that it does not adequately support their objectives. Then,

[1] Unspecified details are to be filled in by the execution agent whose job is to execute the specified process. The execution agent may be a tool or a human. When humans are given the job of executing parts of the process which are not completely specified, they are being counted upon to use creativity and judgement in carrying out these tasks. This is in fact the standard way in which virtually all software tasks are carried out now. Thus customizability enables process designers to assign creative tasks to humans where that seems desirable or necessary, but to restrict such freedom and creativity in other places.

they tune the process accordingly. Thus the environment must be able to *dynamically adapt (evolve)* the process as the need for a new one emerges.[2] However, effective evolution of a running process is difficult as it has to be done in such a way that it does not cause the needless waste of existing software objects which are reusable.

- *Incremental implementability.* Each maintenance process has a set of requirements which should ideally be fully determined before the process and its associated support environment are designed. This suggests that there should be a careful study of the characteristics of the process based on the current and predicted organizational objectives and the levels of available automated support. This is clearly a very long process, which makes the development of the process and environment by traditional methods unrealistic. Furthermore, understanding of any process is usually gained by experiencing it, making it particularly hard to specify it beforehand, and suggesting that it may best be determined dynamically, and implemented *incrementally.*

In summary, the key characteristics that a maintenance environment should have focus on both describing and aiding the process itself in a customizable (programmable), user-tailorable, dynamically adaptable, and incrementally implementable fashion. The need for this sort of an environment is more urgent for supporting maintenance processes than for supporting development because maintenance is currently so poorly understood and so variously perceived. We believe that all of the characteristics described above can be achieved with an environment which is designed around the notion of *Process Programming* [Osterweil 87].

3.2. The Process Programming Environment Paradigm.

Process programming is very much like classical application programming, but in a new application domain--namely software engineering. Each process program describes a single process. The details in this description reflect the specifics of the way in which the process engineer feels that the process should be carried out. Process programs should be developed just as any other program is developed, and, in particular, should go through requirements specification and design phases. The design should invariably be hierarchical, with different hierarchical levels probably being created via iterative refinement. Each level might support a different view of interest and might be designed by a different process programming group. For example, the top level of the process program may be designed by a project manager who may employ some existing process modules. The details of each subprocess may be written by the managers of these subprocesses. Executing these process programs on a computer facilitates the understanding of the status of the project by suggesting that this is tantamount to understanding the state of the execution of a classical application program. Further, the timing and substance of reviews is also suggested by examination of the process code. Here too, the nature of these reviews is not unlike runtime checking of an executing program. Much of this checking might well be supported by automated tools. Indeed, these would appear as subprocesses or contingency handling tasks.

The top level structure of the process program is not necessarily a sequential algorithm. It is best viewed as a collection of cooperating processes working on a shared object-base. The object-base contains and manages such software objects as source code, test cases, requirements specifications and so forth. The process program code, design, etc. are also stored in the object-base so that they themselves can also be maintained as necessary.

Thus, process programs are customizable specifications of how to support and enforce a specific process. It is important to note that this support is not necessarily strongly restrictive in leading the user through the process. Neither does it need to be weakly supportive, as is the case for most classical environments, leaving the user to make all the important decisions and take all of the key initiatives. The degree of support and leadership specified in a process program, and provided by an environment which executes the process program, may vary widely. The purpose of the process program is to specify and enforce only that which the process programmer wishes to specify. Thus the process program is a vehicle for delineating program and management policies. It is also a vehicle for specifying any communication and cooperation

[2] This sort of flexibility is different from customizability, which entails supporting different versions of a process, each of which is fixed and designed before the process starts executing.

protocols that project management may wish to impose between the different subprocesses in a larger process. If the environment contains a large number of smaller subprocesses, then the higher level process programs could incorporate the flexibility to rearrange them differently for carrying out the process in differing ways.

Process programs are true software, so they are themselves maintainable. The fact that they are assumed to have been developed in a systematic way through iterative design refinements improves their maintainability by allowing the users to write new subprocesses and change existing ones. An analyst, for example, might implement a favorite procedure for creating product requirement specifications, or even pick one from an existing process program library. This can then be modified when more experience is gained. This modification is nothing but a "spare parts" maintenance process [Gilb 82] which in turn should be carried out with the support of an existing maintenance process program and runtime environment. These processes are also user-tailorable, provided that the interfaces between them have been carefully specified in the design of the the process, and are not altered by the user wishing to make changes.

The tailorability and customizability of process programs provides a very flexible framework for the incorporation of new tools. "Tools" in the process programming context are thought of as being analogous to operators in classical programming languages. A tool operates on input data to produce some outputs which might be involved in other operations. For example, in the classical software development process, the requirements specification subprocess can be viewed as an operator, which produces a requirements specification as its output. This output is then passed to the test plan development subprocess, which is also viewed as an operator. The output of each operator may be considered to be an ephemeral or persistent object to be managed by the object store that underlies the environment. Both the operands and the results of the operators are considered to be instances of types defined in the process program. Incorporating a new tool into a process programming environment is supported readily as a consequence of the environment's tailorability.

There are other advantages in looking to the process programming paradigm as a way of meeting the critical requirements for our powerful maintenance environment. Since this paradigm is based on the classical notion of programming, much of the experience and intuition gained over the years in writing and studying classical programs are applicable to process programs too. In addition a formidable programming technology, including structured programming, systematic design, formal verification, and static and dynamic analysis are now also useful in effectively developing high quality process programs. Most significantly, the process program is the same sort of object as the products it supports. This means that a process programming environment could be used for supporting itself. This support of the development of a process program does not require a special purpose environment, but rather an environment of the same sort as the one needed for the products it supports.

Although a process program is very much like the products whose development and maintenance it supports, they are different in their levels of abstraction and hierarchy. We define a *Process program* as the "static description of how the process itself could be carried out, incorporating the appropriate and necessary tools and object base". Such static description and tool support for the process provides a suitable process environment. *Active process programs* are running process programs whose executions are being supported by *process programming environments.*

Process programming environments are also good vehicles for recording knowledge and experiences gained in the course of carrying out software projects. This knowledge can be captured in the form of various statistical data objects describing and abstracting the way in which process execution has proceeded in the past. These objects can then be considered to be outputs resulting from the execution of the process. They can be readily captured within the environment supporting execution of the process, and might then also be used as inputs to subsequent executions of the process. In this way they become bases for experimenting with the process and for improving it -- perhaps even dynamically.

Finally, it is important to note that the characteristics of incremental implementation and dynamic adaptability, which are natural characteristics of process programming environments, make these environments particularly effective in supporting maintenance.

3.3. A Maintenance Process Program.

In [Gamalel-din 88] is shown a complete simplified process program which combines some of the maintenance activities and processes described in section 2. The purpose of this example is to clarify and demonstrate some of the concepts and advantages of process programming environments. It also shows how a process program serves not only as a process documentation vehicle but also as a vehicle for process improvement. A snapshot of this program's execution is given below.

As noted earlier, process programs are software too and, hence, should be developed using a systematic development process of their own. Thus, requirement specification, design, testing, and maintenance are parts of a process program lifecycle. To carry out example further, we indicate below some of the products of some of the development phases for one plausible maintenance process.

3.3.1. Requirement Specification.

A hierarchical description of the requirements for our maintenance process is shown in figure 1.a. This diagram, although incomplete, shows a hierarchical function decomposition of the process. The two main branches correspond to two main maintenance perspectives -- managerial and technical. One branch requires that the maintenance process be carried out by a well organized team. Another branch represents the steps a "change" process must follow -- namely understanding, modification, and revalidation. It also requires the existence of an object store/manager and shows some of its roles.

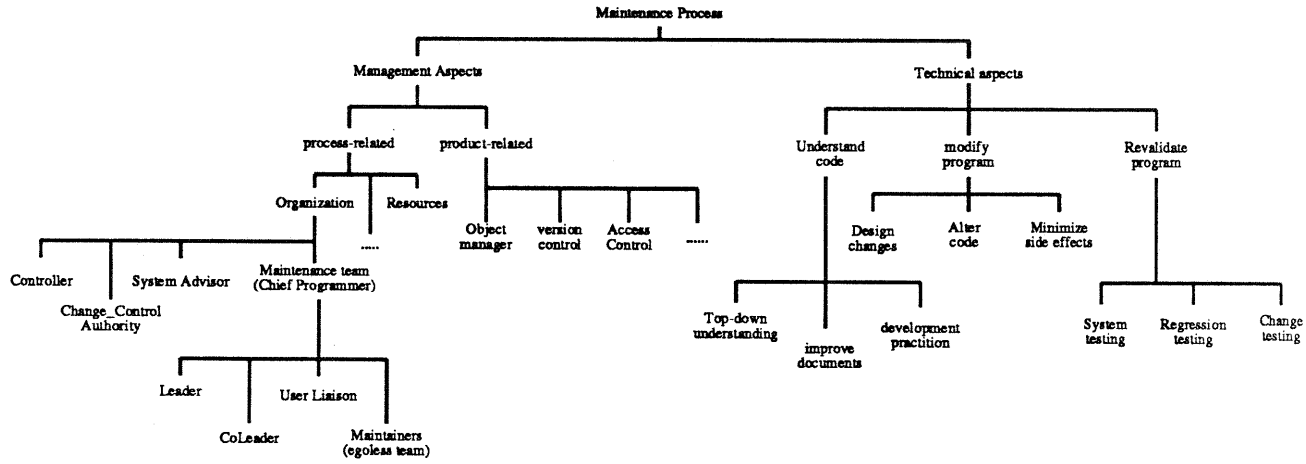


Figure 1.a. A Maintenance Process Requirement Specification
(A hierarchical functional decomposition)

A possible representation of one of the elements of this requirements specification is given in figure 1.b. This particular element specifies the different responsibilities and roles of a maintenance-controller as

```
requirement_element   maint_controller  is
        description:
                It forwards each maintenance request to the appropriate
                system supervisor and provides high-level management for
                the maintenance process. It is assumed to be permanent and
                unique (i.e. one controller for all maintenance activities)

        Education:
                Minimum School years.
                Tutorials/Courses.
                Degree.
        Skills :
                Specify skills needed here.
                Years of experience.
                Domain knowledge.
        function_specs:
                function:
                   high_maint_management:
                        TECH_REQUEST or TECH_QUERY --> DECISION or ACTION;
                   create_maint_team : response time = T1 :
                        CONF_OBJ X MAINT_PLAN  -->  MAINT_TEAM;
                   outside_world_comm : response time = T2 :
                                -- T is a function.
                        user_request or high_manage_request  pass it to
                                maint_organization &
                        maint_team_request  pass it to user or high_manage;
                   job_schedule : response time = T2 :
                        MAINT_PLAN X SPR_TYPE X MAINT_JOB_READY_Q  --> MAINT_JOB_Q;
                   accept_validated_software: response time = T3 :
                        CONF_OBJ --> Boolean;
                   determine_release_criteria:  response time = T4 :
                        CONF_OBJ X MAINT_PLAN X SPR --> RELEASE_CRITERIA;
                   coordination_of_personnel :
                        TECH_REQUEST or TECH_QUERY --> DECISION or ACTION;
                   maint_record_keeping :
                        MAINT_RECORDS X MAINT_PROCESS X Statistics --> MAINT_RECORDS;
                   configuration_control:
                        CONF_OBJ X MAINT_PROCESS --> CONF_PROCESS;

                  communication:
                        two_way_communication with :
                                user, high_management, Change_Control_Authority,
                                System_supervisors, Maint_team_leader;
end requirement_element ;
```

Figure 1.b. A requirement element structure.

well as the other people with whom he/she can communicate. It also describes the education, skills, and experiences required for a person to play such a role. This requirement element example indicates that even humans' responsibilities as well as capabilities are specifiable as part of the process program requirement specifications. The example described here makes frequent reference to the software object, "Job schedule" to show how to program a human responsibility. Its signature indicates that it employs maintenance plans, system problem reports, and an ordered list of available maintenance jobs in order to schedule new jobs. The time limits assumed for scheduling jobs according to a specific request is given by the time function T1. All other nodes of the hierarchy of figure 1.a can be defined similarly.

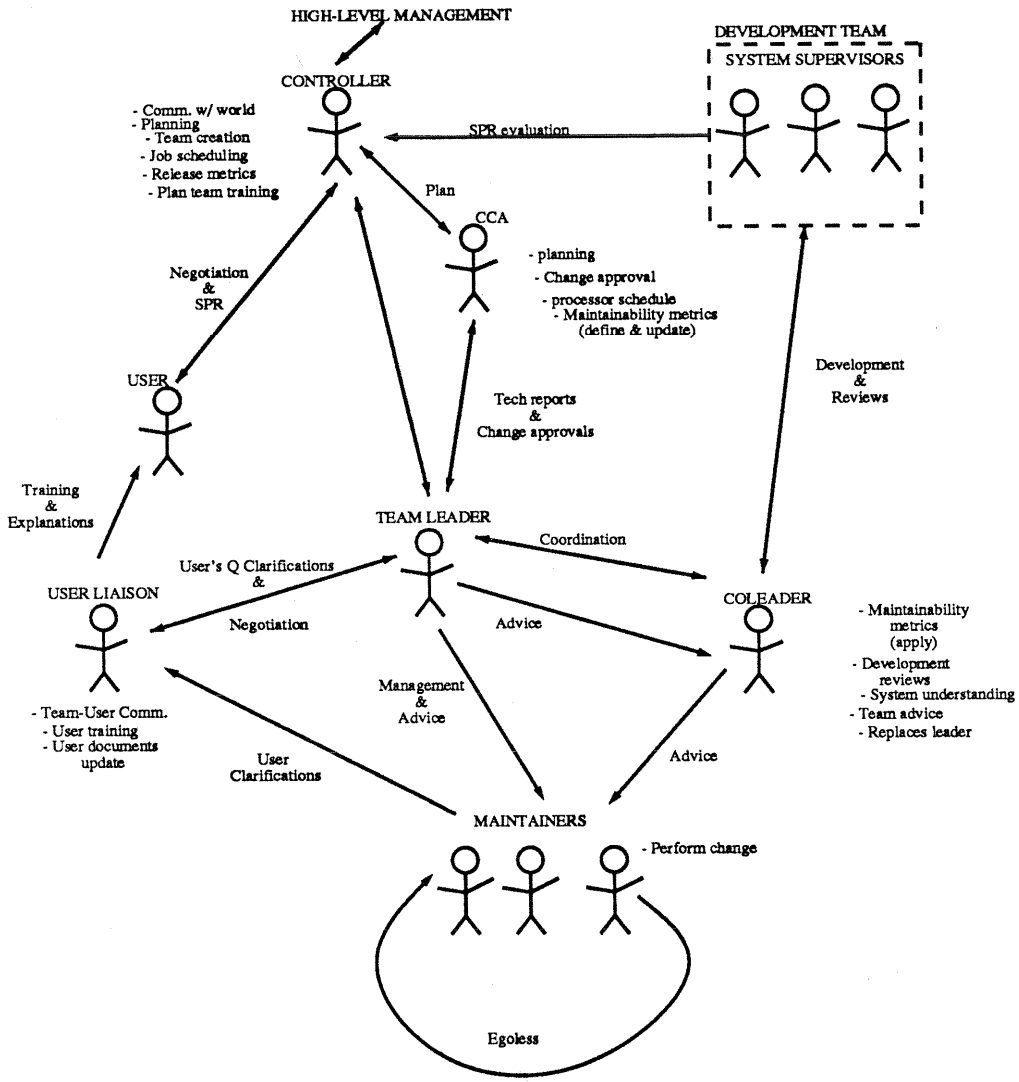


Figure 2.a. The Communication between and Authorities of
The Maintenance Organization members.

3.3.2 Architectural Design.

Focusing on the managerial branch of figure 1.a, we note that figure 2.a and figure 2.b are examples of specifications of key documents. Figure 2.a is a representation of the maintenance organization. It describes the responsibility of every individual of the team and some possible messages and communication media among them. Figure 2.b, however, shows the flow of events.

Some of the personnel roles as described in figure 2.a are not relevant to code changing activities. However, they are essential to improving the overall process. Team training and maintainability metrics definition are examples. Team training is another process which is required to improve maintenance quality. Metrics and design guidelines must be available to the development team in order to produce highly maintainable code. A maintenance team representative (Co-leader) must be present in development reviews. This shows how a maintenance process program may impose extra requirements on other processes and how they interact. However, other roles are integral parts of the maintenance process. Job scheduling is one of these. We note that the process specification indicate that when this subprocess is done, scheduling information will be passed to the "Change Control Authority" who initiates and manages job-team assignments.

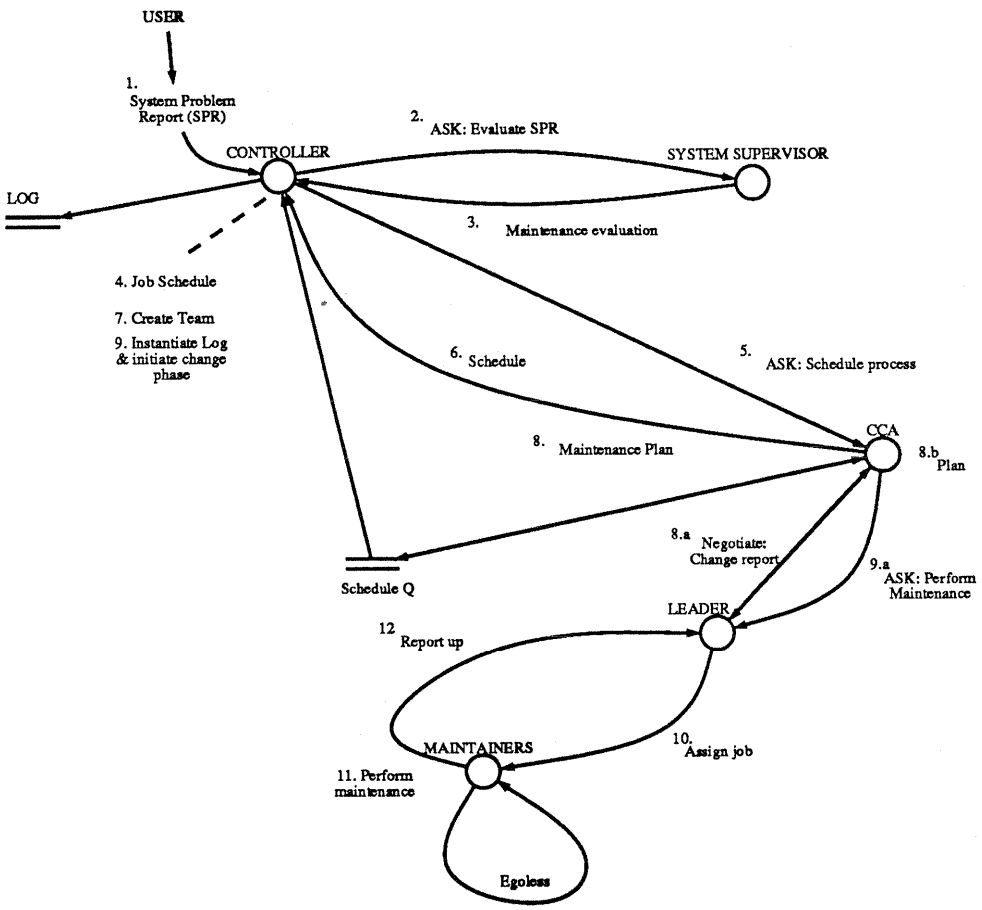


Figure 2.b. Flow of Events inside the Maintenance Organization.

3.3.3 Detailed Design.

As is shown in figure 2.a, maintenance job scheduling is one of the responsibilities of the "Controller". This activity may be programmed as a procedure (see figure 3.a) which must be executed by the "Controller."

```
job_schedule (rep : SPR, eval : Maint_evaluation_report) ==>
        -- this method is to schedule the maintenance process according
        -- to decisions made with the CCA. Scheduling is based
        -- on a multiple priority queue mechanism commonly used in
        -- operating systems. Four queues are used. The highest priority
        -- is assigned to the first Q while the lowest to the last one.
        -- each Q is ordered according to the priority assigned to each
        -- of the contained processes.

        local log : Maint_log ;
        if (not is_cost_effective (rep, eval) ) then
                -- it is not cost effective, so reject the request.
                -- this is based on the supervisors evaluations.
          resume_old_operation; -- do what was being done before.
        end if;
        case eval.maint_type do
          CORRECTIVE :
                if (eval.prelim_priority <= SEVERITY_LIMIT) then
                        log := self.start_maint_plan (rep, eval);
                          -- asks itself to start another activity.
                        load_into_Q (maint_request_Q[0], log);
                          -- This will load into the Q #0 sorted by the
                          -- priority.
                        self_send (to= CCA, info= NULL,
                            for= processor_scheduler);
                        resume_old_operation;
                else
                  -- not severe : catalog for error correction.
                        log := (log)(eval, rep) ;
                        load_into_Q (maint_request_Q[1], log);
                        self_send (to= CCA,   info= NULL,
                            for= processor_scheduler);
                        resume_old_operation;
                end if;
          PERFECTIVE, ADAPTIVE :
                if (eval.prelim_priority <= PRIORITY_LIMIT) then
                        log := self.start_maint_plan (rep, eval);
                        load_into_Q (maint_request_Q[1], log);
                        self.send (to= CCA,   info= NULL,
                            for= processor_scheduler);
                        resume_old_operation;
                else
                  -- not high : catalog for mid scheduled development effort.
                        log := (log)(eval, rep) ;
                        load_into_Q (maint_request_Q[2], log);
                        self.send (to= CCA,   info= NULL,
                            for= processor_scheduler);
                        resume_old_operation;
                end if;
          PREVENTIVE :
                        log := (log)(eval, rep) ;
                        load_into_Q (maint_request_Q[3], log);
                        self.send (to= CCA,   info= NULL,
                            for= processor_scheduler);
                        resume_old_operation;
        end case;
end_procedure  job_schedule
```

Figure 3.a. A detailed design of "job_schedule" activity.

If an object-oriented programming language were used to do this, for example, "Controller" would be considered to be a job and maintenance job scheduling would be one of its methods. Procedure "job_schedule" presents a plan for chronologically ordering maintenance jobs on priority basis. Job priority is determined as a function of its maintenance class (corrective, perfective, or adaptive) as well as its severity, cost, and resources availability.

```
--
--                  System Problem Report
--
-- System Problem Report: It is issued by the maintenance requester, e.g.
--    the user, or the maintenance organization as a step in a preventive
--    maintenance plan. Some of its data are filled in by the requester while
--    others are filled in by the maintenance group. Such report is the one
--    that triggers the whole maintenance process.

type  SPR  is  record  of
      system : System_id;
      spr_log : string ;
      version : Version_id;
  -- data filled in by the maintenance group for control purposes:
      log_no : string;     -- unique and assigned by CCA.
      log_date: Date ;     -- the date it is logged by CCA.
      problem_with: Config_component;
                        -- database, document, routine, module
      status : (REVIEWED,  -- reviewed to determine the appropriate action.
            ASSIGNED,   -- assigned to maint. group to fix.
            FIX_AVAIL,  -- fix is done and tested but not integrated.
            INTEGRATED, -- integrated and tested .. test not finished.
            SCR_REJECT, -- integration test failed.
            SPR_CLOSED, -- passed the test and problem is solved.
            POSTPONE)   -- it has low priority and not to be handled now.

  -- data filled in by the requester:
      originator:          -- information about the requester (name, addr)
      time :  Date_Time;        -- of problem discovery.
      activity : (SYS_TEST, INTG_TEST, DEVELOP, OPERATION, ENHANCE, ...);
            -- the problem discovery activity.
      probl_description : text; -- description of either error or enhancement
                        -- required.
      request_fix_time : Time;  -- latest time to fix the problem.
      input : File;             -- the input used when errors are discovered
                        -- Or a test case (input case) which is to be
                        -- supported by the enhancement.
      output: File or Printout; -- used in case of error. or a proposed output
                        -- given in the form of a document.
      memory_dump : Dump;   -- used only in case of error.
      HW_descr  : HW;           -- description of the hardware used or needed.

end record
end type SPR;
```

Figure 3.b. A data structure definition of SPR type document.

Documentation and document layouts are related to each other in this process program in the same way as data instances and their corresponding type declarations are related in traditional programming. System problem report (SPR), as an example, is one of the inputs to "job_schedule" procedure. It is an

important data element to be considered in Designing the process. Figure 3.b shows a structure representing the SPR document template. It also specifies who is responsible for providing each of its data fields.

The above example shows that it is possible to specify and design a maintenance process as we do other software. It also shows that humans are essential components in the process and hence their roles and limitations are subject to definition and design. It should now also be clear that any such programmed activities may be changed, adapted, and tailored to address new requirements because they are programs. This boils down to nothing but maintenance activities applied to the process program itself. The next section is a more detailed discussion along this line to explain how process programming improved our understanding and characterization of maintenance.

3.4. Process Environments and New Maintenance Perspectives.

The notions of process programming and process environments open new research dimensions and areas. We believe that two of the most important new dimensions are process maintenance and dynamic maintenance.

3.4.1. Process Maintenance.

A software product may be maintained in different ways and so may the process by which it was developed. To help us to discuss and reason about this idea, we define *product maintenance* to be "the process of maintaining classical programs along with their configurations." Whenever it is necessary to modify such a classical program or its configuration, a maintenance process should be developed and applied. Such a process should be formally specified as a program that takes in both the classical program and change request specifications and produces a new version of the classical program which satisfies the change request specification.

For the sake of clarity and differentiation between these two types of programs, we will refer to classical application programs, together with all their associated designs, documentation, testsets, etc., as *products.* Specifications of the processes by which such products are developed and maintained will be referred to as *processes.*

Another way to maintain a product is to maintain its development process program. Before elaborating on this new type of product maintenance we first define *process maintenance* to be "the process of maintaining process programs in order to either change the process itself or the products it produces." This definition implies two reasons for maintaining a process. A process may be maintained for the sake of its own improvement, to meet new organizational objectives, or to adjust to the availability of fewer resources. On the other hand, product quality, structure, and other characteristics may also be reasons for changing the development process. The latter is what we refer to as *product-related process maintenance* which, more precisely, is "the process of maintaining process programs in order to maintain the products they produce."

Product-related process maintenance offers a number of important advantages. When one alters an existing process program one has the opportunity to specify how products currently specified as intermediate results of the the execution of the current process might be created differently, be replaced or reused in an improved processes, leading to an improved product. The maintainer has the advantage of evaluating which products and which subprocesses have proven to be worthwhile and reusing them, while also being able to augment and enhance them.

We observe that what has just been described is in fact just a formalization of the way in which much maintenance is already being done. Usually one is faced with existing software which needs improvement. Sometimes it is decided that the software is in need of improvement or enhancement because the process by which it was built is considered deficient. For example, it might be that the software has not been exposed to an adequate testing process. For whatever the reason, it is then often decided that the process itself needs to be maintained to improve the product by improving the process. After being maintained, the process is employed to produce other products which have important similarities to the products produced before maintenance, but which have distinct differences as well. We refer to this kind of maintenance as *static maintenance,* which we define to mean "modifying a program representation in its static state." Again

both products and processes are subject to this kind of maintenance. To distinguish between them we use the names *static process maintenance* and *static product maintenance.*

3.4.2. Dynamic Maintenance.

Consideration of the problems posed by thinking of process programs as being executing software leads to a new and different type of maintenance--namely dynamic maintenance. *Dynamic maintenance* is "the process of modifying a program while it is being executed." Although, conceptually, both products and processes can be maintained dynamically, we prefer to focus our discussion only on *dynamic process maintenance,* maintenance whose subject is a process rather than a product.

The need for this new type of maintenance becomes most apparent when considering the role of humans as execution agents for subprocesses of process programs. When humans participate in the execution of a process, they usually follow a specific plan, which we call a process. This plan may be documented in some manual, or guided by a computerized process program which assumes that users will interact at specific points, providing inputs given in a strict form and in a strict sequence.

The integration of humans as execution agents for subprocesses in process environments is a key difference between process programs and classical application programs. Process programs are, then, formalizations of long term processes which encompass learning and exploration activities. Iterative and continuous improvements to processes are to be expected, even during the course of execution of the process. The process itself is subject to *incremental implementation* and so is the process environment. Thus, dynamic spare parts maintenance is obligatory here.

Dynamic back-bone maintenance is also expected for process programs. During the long term course of executing process programs, new tools might become available, design changes and organizational rearrangements must also be expected. These necessitate dynamic change of the running process to accommodate the change.

In introducing the concept of dynamic maintenance we believe we are adding a new dimension to the concept of maintenance along with a new set of problems. Before discussing such problems, we address another key issue that makes process programs far more complicated--namely the persistent objects with which process programs deal.

Process programs are meta-programs which are aimed at developing and maintaining classical programs. The operands of their expressions are large, complex, persistent objects, such as source code, and requirement specifications. The operators are procedures and tools which, for example, may require evolution. Let us consider, for example, the process of creating requirement specifications. The product of this process is a structure of requirement specification elements which may be created, for example, using some fixed template. Some consistency checks may be applied to every requirement element upon its creation. After the process has been running for a while, changes may be desirable. Modifications like changing the template structure (i.e. the description of the type of the persistent operands), the consistency rules (the semantics of a tool or operator), or even the requirements specification itself (for example by adding extra relations among the different requirement elements), are all conceivable. Such changes are not just changes to the process program in the sense of classical static maintenance. Already existing persistent elements must also be updated in accordance with the new changes to guarantee consistency. We define *execution history maintenance* to be "the operation of updating the past execution states of a program before accommodating a change so that the current state will be the same as if the program were reexecuted starting at its beginning."[3]

In dynamic maintenance employing an execution history update the current state of the program must be altered to bring it to the state which would have been attained if the changed program had started re-executing from the beginning. In the case of classical programs this is generally achieved by reexecution

---

[3] History maintenance causes maintenance in process programming to differ from that in real time programming, where it is unnecessary and impossible to update the history of execution. In real time programming, although the program is maintained during its execution, the change begins its effect on the state of the program starting at the time of confirmation of the change. The change only affects future states, based on the current state, because this type of program does not deal with persistent objects as process programs do.

of the program. However, in long execution process programs, that is infeasible. Re-execution means that the process will never terminate as it continues to evolve. Further it is impossible to reproduce the behavior of humans as they support the reexecution of the processes in which they participate.

A product can be maintained in either way. In the classical method, the formalisms by which the product is specified and the relations among the different objects of the product are all considered key to guiding maintenance. Alternatively, *product-related process history maintenance* is an efficient and more accurate way to carry out product maintenance. It is defined as "the act of maintaining a product by dynamically maintaining its development process's execution history."
So, in product-related history maintenance, the development process is the key. Even if the process had been terminated and its product had been already delivered, the product could be maintained via its development process. Maintaining the history of the reactivated process by changing the values of some of the persistent objects it deals with, is a more appropriate and efficient method for maintaining products. The former approach does not encompass any history maintenance and hence does not require much support. However, it requires much more effort to provide the same accuracy and guaranteed consistency of the changed components according to the development metrics.

The notion and proposed techniques of dynamic maintenance seem to have applicability beyond their use in supporting process environments. In program debugging, dynamic maintenance could be used to restart the debugging process after a program has been modified. The effect would be to enable the resumption of the debugging run without having to reinstrument, recompile, reload, reexecute, and re-establish the debugging environment following the change. Product-related history maintenance is also a useful technique in supporting dynamic updating of the schema and the database in a database application system.[4]

The newly introduced maintenance notions of process and dynamic maintenance as well as the related topics of history maintenance and dynamic backbone and spare parts techniques, all combine to greatly complicate the activities of maintaining process environments themselves. This raises the need for maintenance environments and tools to support these new complicated activities. Section 4 proposes a powerful maintenance environment and tool to overcome the complications these new notions introduce.

3.5. Yet another software life cycle.

The process programming perspective can also be used to suggest a fresh perspective on the software lifecycle. Examples of classical software lifecycle paradigms are waterfall, continuing evolution, and "b" models [Birrell 86]. Figure 4 shows a proposed new software lifecycle paradigm -- namely the "process-oriented software lifecycle" paradigm. Most classical paradigms describe how software develops from its earliest conceptualization and specification, and going through evaluation, analysis and maintenance. We believe it is important to consider that software development really begins with the development of the process which is to be used to develop the product. This process itself has its own lifecycle, but we believe that it is essentially the same as the lifecycle of the products that it produces. Thus our new lifecycle paradigm must be able to represent different levels of software lifecycles -- the product, the development process producing the product, and the metaprocess of producing the development process itself.

Figure 4.a depicts a static view of this process lifecycle. It suggests that a process has to be fully developed (defined) before it starts being used for product manipulation. Such manipulation must be completely specified in the process program but it may specify either the entire development process or any of its principal subprocesses (e.g. requirement specification, design, or even maintenance). Once the process is fully developed, it can execute to achieve its purpose. However, its execution is probably spawned by a higher metalevel of the lifecycle--namely the product cycle. The results of executing the process may be monitored and inspected for the purpose of evaluation and evolution. Evaluation results or the need for product or process evolution may then cause the cycle to evolve.

---

[4] Theoretically, efficient history maintenance techniques might be used to rebuild the database faster as they would only update selected entities. Pragmatically, these efficient modifications may cause inefficiencies at runtime, due to the lack of adequate consideration of clustering and disk allocation issues. They may counterbalance initial gains and hence need careful consideration.

This view is consistent with our earlier discussion of product, process, and product-related process types of maintenance as defined in section 3.4. It thus shows that the development cycle is not different from the maintenance cycle.

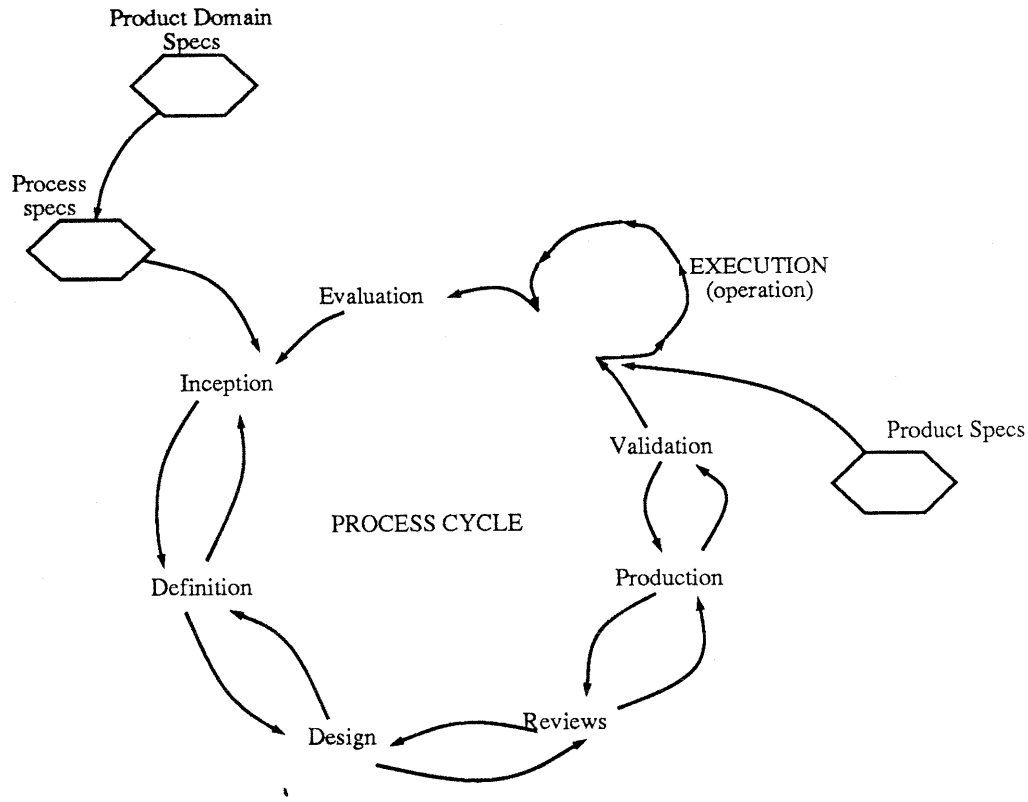


Figure 4.a. A Static View of the Process-Centered Lifecycle.

This static view is too limiting, however. The process and product development cycle we have described is highly dynamic. Further, consideration of this dynamic nature suggests a way in which we can think of the development process as a maintenance activity. That is, by widening our understanding of the maintenance process and improving its model to capture all of its dynamism, one can think of the maintenance process as a super class of the development process in a hierarchy of processes. To explain, consider the incremental implementability and dynamic adaptability of processes which are maintenance activities, yet essential parts of process development. Figure 4.b is an abstraction of a lifecycle representation including both the dynamic and static parts of the cycle. The detailed diagram of the dynamic view of the cycle is shown in figure 4.c.

An executing process program will continue until finishing its task or until an external maintenance request is encountered. The latter initiates the dynamic cycle while the former leaves the process in an idle state. Once idle, the process waits for a request for either a static or a dynamic maintenance activity or even for a request for a product redevelopment. A static maintenance request initiates the static cycle while a dynamic request initiates the dynamic one. A new product specification causes the reexecution of the process program until completion or any maintenance interruption.

The dynamic cycle starts with either a process (e.g. adaptation, incremental implementation) or a product maintenance request (product-related history maintenance). The latter can even happen while the process is idle in order to maintain a delivered product. In such a case, the process is reactivated and a whole dynamic cycle is initiated for the sake of maintaining the appropriate execution history. It should be noted that the activities of the dynamic cycle are similar to those of the static one but are done incrementally. The dynamic cycle terminates by updating the execution history. The process, then, resumes its previous state -- execution if it was interrupted and idle if product-related maintenance was requested.

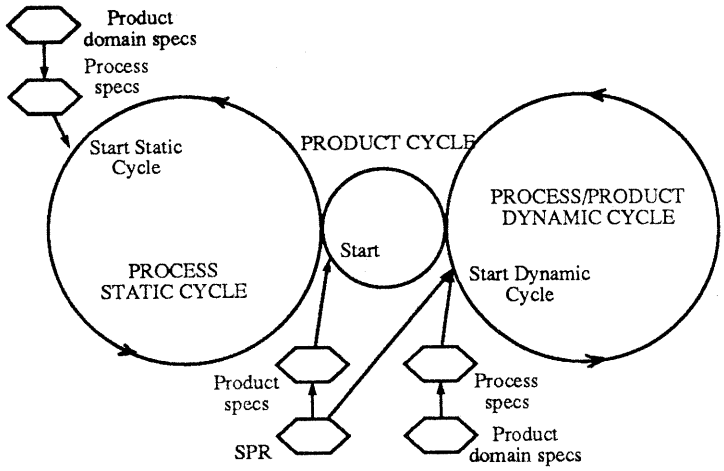


Figure 4.b. An Abstract of the Process-Centered Software Life Cycle Paradigm.

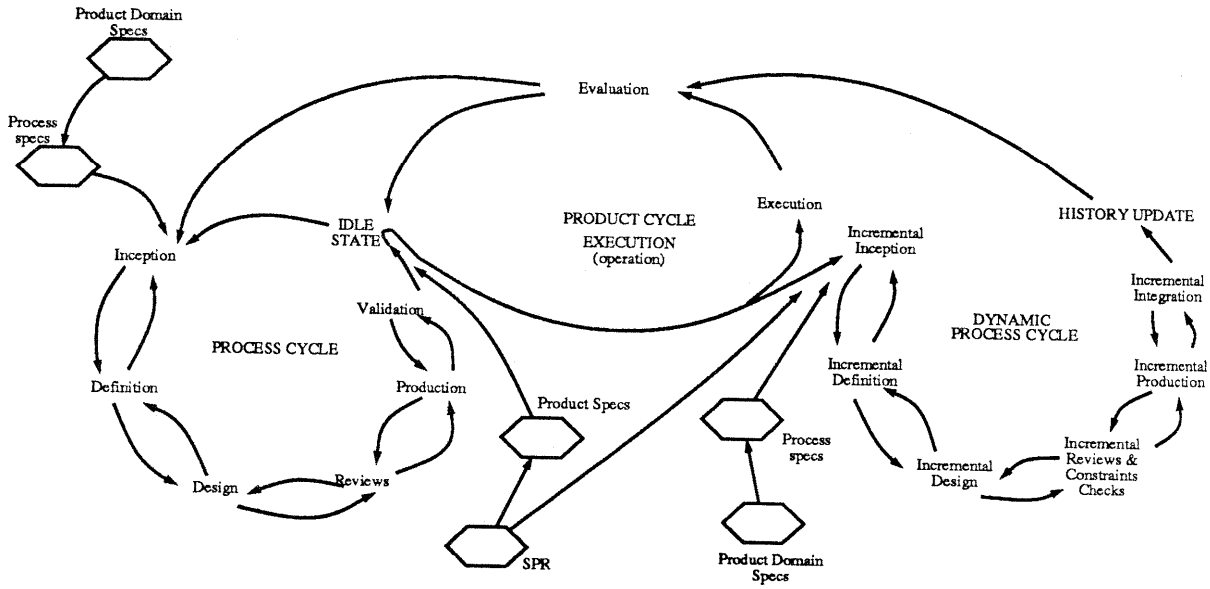


Figure 4.c. The Process-Centered Software Life Cycle Paradigm.

4. A Maintenance Meta-Process Environment

Software process environments, which support the software processes during the course of its execution, are software too and, hence, they themselves need support. The dynamic evolutionary nature of process programs, however, poses difficult problems, and raises the need for dynamic maintenance.

4.1 The Unique Characteristics of a New Maintenance Process Environment.

Environments for supporting the maintenance of process programs will share some of the characteristics of environments for classical software, but will also have to satisfy important new requirements. For example, where classical environments might be quite useful even if they support only static maintenance, it is essential that a process programming environment support both static and dynamic maintenance. In attempting to meet these harder requirements, process programming environments will have to incorporate some new tools that will be challenging to develop.

For example, the activity of coming to understand programs should be automatic and powerful, and it should respond to user queries rapidly. A whole new set of analyses is required in order to support rapid reply to queries in the context of dynamic maintenance. Query response must be based upon the analysis of execution histories which must be kept up to date as execution of the process proceeds. In addition, we expect that different types of users (eg. workers and managers) will need to pose queries, suggesting that the responses might have to be different as well.

New and difficult tools must also be developed to support the subprocess of assessing and evaluating changes. Before confirming proposed changes, they must be shown to be correct and consistent with other program structures and software objects. For example, changing the type of one field in the template of a requirement element type definition may be relatively straightforward in a static situation. The change is made, and affected programs are then recompiled and rerun. In a process programming environment, however, one must expect that this sort of change will have to be made dynamically--as the process program is executing. In this case it is necessary to re-evaluate the program code involved in producing that field as well as code which is involved in relating it to other software objects. The execution history must also be reevaluated to see if changes must be made to previously created persistent instances of the requirement element type. Clearly it would be far easier to simply rerun the process program after such a modification, but this is usually impractical or impossible. Thus a process programming support environment should incorporate a facility for dynamic *re-evaluation*. Re-evaluation differs from re-execution in that it simulates the execution of only those statements which have been affected by changes which have taken place. This is a new sort of optimization which is necessary in order to assure that dynamic maintenance can be done rapidly.

The essence of re-evaluation is to assure that changes are consistent with the syntactic and semantic constraints of the programming language as well as the pragmatic constraints imposed by the nature of the problem. Language syntactic constraint violations are relatively easily to detect statically. However, many semantic and pragmatic constraints are hard if not impossible to evaluate statically because of their dynamic nature. This suggests that *dynamic constraint checking* (eg. by means of assertions) *dynamic constraint propagation* are key techniques in dynamic process program maintenance. It must be noted that even these dynamic checks can not guarantee absolute correctness, as they can at best only assure consistency and correctness up to the current point of execution. Proposed changes might still set up inconsistent or incorrect behavior later on. Thus still further complex tool support is suggested.

A key capability of the tools and processes needed to support dynamic maintenance is ripple effect analysis. Ripple effect analysis is essentially a recursive operation, which relies upon representations of the interdependencies of the various program objects and those objects which have been changed. Some changes may be entirely local, and have no direct effect on other program entities, while others may propagate widely to eventually affect many other entities. Static ripple effect analysis seems to rest upon relatively better understood principles and structures, but dynamic ripple effect analysis seems far harder, and seems to require the need for analyzing and adjusting execution histories. This seems to require the development of difficult new tool technology and raises the prospect of unexpected and unfamiliar outcomes. For example, the result of changing the value of an object at some point in the execution history may cause a ripple effect leading to the re-evaluation of other variables and the eventual re-evaluation of flow of control predicates which may then cause the execution of a different path through the program.

Thus the environment must detect when re-evaluation has begun to cause execution of a new path, and then roll back to the deviation point, and restart execution from this point. It is even possible that a given change may cause a new execution path which criss-crosses the old execution path, raising the possibility of different rollback points. A specially devised algorithm may be needed to select a single optimum roll back point from which to resume execution. Such a point might not necessarily be the earliest in the program execution history. For example, when the deviation leads to an alternative branch in an if-then-else construct where the two branches differ only in the value assigned to the same persistent element, reexecution could proceed beyond this deviation point, since re-evaluation could set the proper new value.

Coarse grain module inter-relationship analyses are excellent supports for ripple effect detection in classical static maintenance. Finer granularity is required in the case of dynamic maintenance, especially for process programs, since this maintenance may be request by users who are working at different metalevels and different levels of abstraction. Thus, what is considered to be an operator under some circumstances, might be a process at another level. For example, process program maintenance may necessitate the replacement of a tool (operator) by another which may have either the same or different semantics. Similarly type definitions must also be expected to change periodically during the execution of a process program. Consideration of the need to make such substantive changes during the execution of process programs, lead us to decide that the basic elements of programming languages must be the subjects of ripple effect analysis in a full process programming support environment.

Retesting is another very important activity in the maintenance process. In classical static maintenance, regression tests, dynamic debugging, static analysis of the changed program, and coverage tests are methods used for assuring correct changes. Those techniques are not adequate to support dynamic maintenance, since dynamic maintenance is carried out during the actual execution of the process. Other compensating techniques such as history maintenance, execution monitoring, and dynamic constraint checking are needed.

Another requirement of a process programming maintenance environment which is particularly difficult to satisfy is that it should be able to support maintenance of software objects developed using different languages. Even for a single software product, we must expect that various of its component objects, e.g. the requirement elements and design specification elements, are legal strings in different languages. Each language has its own syntax, semantics, and constraints upon which similar maintenance analysis could be done. Hence, ideally, the maintenance environment should support maintenance of software objects expressed in any language, given the language definition. This last characteristic imposes severe restrictions on the design and implementation of the environment and on all of its components and tools. The environment components must be built in such a way as to be language customizable. That is, general tools must be built so as to employ the given language specifications for customizing generically built tools (eg. by using *operator overloading*.)

In summary, the characteristics of a process programming maintenance-based environment requires quite a different approach to and perspective on the maintenance process as a consequence of the dynamic nature of the programs it is supporting. Among the essential features of such an environment are rapid response to both static and dynamic history analysis queries as well as quick and consistent (error-free) dynamic changes for all maintenance affected program components including the execution history. Ripple analysis tools, re-evaluation techniques, dynamic constraint propagation, operator overloading, and incremental development are some useful tools for such an environment.

4.2. A Proposed Maintenance Environment.

A process programming environment capable of supporting both the development and execution of process programs must be capable of supporting their continuous static and dynamic evolution. Such a support environment must thus take the form of a maintenance meta-process environment with facilities for user interaction. Appendix 1 gives an example of such a process program. It should be noted that this process takes care only of the technical issues of the maintenance process. Management issues are more familiar, being very similar to those used in traditional static product maintenance.

4.2.1. User-System Interaction Scenario.

Let us consider a process program for developing a structured software system, where this process is running under the support of a maintenance meta-process programming environment. The meta-process is initiated when the user watching and evaluating the progress of the running process discovers some inefficiences or defects. In order to fix them, the process program must be quickly and carefully studied in order to determine the requirements for, and design of, the improvements.

Two types of support should be provided -- one for the "program-understanding" activity (e.g. an interactive query answering facility) and another for explanation, reasoning, and guidance activities.

Some examples of analytic data that is needed in order to help answer users' queries about the subject process program are:

- data and control flow graphs, module inter-relationships, dependency graphs, and so on;

- symbol tables (e.g. for variables, procedures, flow graph branches, ...etc.) which contain static and dynamic information such as; types, occurrences, dependencies, aliases, evolution graphs, variable roles and goals, searches for patterns, and so on; and

- execution histories for both the program and its individual entities, where these histories incorporate version depictions based on the previous changes applied to the program to affect its history. This information helps the maintainer to understand the decisions and rationales for changes and simplifies re-do and undo operations.

The explanation and guidance support facility should provide more information, based on program semantics and problem pragmatics. Feedback on operations and tools usage, constraint rules, and goals of code chunks are examples of such guidance. Another example is information on traces through the execution history and source code to help in answering questions such as "why did we get this result?" and help in providing guidance on "how to do so-and-so."

Such explanations and code understanding facilities are vital to the process engineer in designing needed process modifications. Since there is no feasible way of retesting process changes, all changes must be carefully studied and evaluated before being confirmed and adopted into the running process. Thus, the maintenance meta-process must derive from the change request a set of static and dynamic consistency checks, not unlike the static and dynamic sequencing constraint checks which a compiler makes.

The subprocess of actually making a change begins when the maintainer specifies the change. Even this is relatively complex/ This change request must specify not only the "change" but also the textual points at which the change is to be made, and the execution state at which that change is to begin its effect. Such a state may be either a specific single state in the past history, or it may be at all of the history states, it may be the current state or it may be a future state, in which case there will be no effect at all on the history. Hence, the dynamic checks must apply not only to the current state of execution but also to all the affected history states.

The ripple effect of the change must then be assessed from both static and dynamic views. The static and dynamic ripple effects are quite different from each other, and both must be studied. An object is called conditionally dependent on another object when the evaluation of one depends on the value of the other only if the program execution took a specified control path. In the dynamic sense, not all statically dependent objects are dynamically dependent, and the same is therefore true of the corresponding ripple effects. In other cases, two objects might dynamically depend on each other in different ways depending on the differing effects of different execution paths. The dynamic ripple effect analyzer must work correctly under all of these circumstances.

All such ripple effect analyses must then be collected and constraint checks must be recursively carried out. Some of these ripple effects can be handled automatically while others can not. In addition, some changes have only dynamic effects while others have static effects. For example, changing the type of an operand for an overloaded operator may have no static effect while this will affect the value of the operation output, and hence history maintenance must be activated.

The change and all of its ripple effects must then be reported to the user for confirmation. Once confirmed, the set of changes then become permanent and the appropriate history maintenance is initiated. For a change to become permanent, all the program representations, including those which are used for

analysis and ripple effect detection, must be incrementally updated to accommodate the new change.

The re-evaluation technique should be used to speed up the process of history updating. History maintenance results in the creation of a new path which then has to be added to the history graph (each path represents a sequence of execution states). This new path starts at the first affecting state of execution and is associated with a list of the requested changes. Such a representation is helpful in carrying out an undo operation in case of future failures due to that change. Finally, the execution pointer must be set to the proper point in the program being maintained.

Different statistical information such as the number, type, cost in terms of the ripple effects, and nature of the changes should be collected to help quantify the maintainability of the process and to help measure efficiency.

The above scenario sketches a sequential process which supports the maintenance of an active process environment, and shows what activities are needed for proper support. In the next section we briefly discuss some of the powerful operators (tools) which are needed to support this process.

4.2.2. A High Level System Structure.

Figure 5 shows a high level structure and data flow diagram for a proposed maintenance meta-process programming environment, capable of supporting the process program of appendix 1.

It should be noted that this environment makes important use of dynamic binding. The tools are assumed to be bound to their meanings at application time. The semantics of each component is defined via two sets of rules. The first set of semantic rules is implicit in the tool itself, e.g. a static analysis tool has the implicit semantic of "statically analyze a program" with no information about the language used. The second set of semantics are imposed via a set of rules which specify the language which the tool must be able to understand. Such tailorability gives the proposed maintenance process programming environment the flexibility to use the same tool capabilities for different purposes. Although this overloading is not always recommended, it seems particularly appropriate for process programming. Due to the continuous emergence of new types and dynamic typing, a single process program may need to handle different classes of objects in similar ways.

The following is a brief description of the major components and tools of a maintenance environment as sketched in figure 5. Appendix 2 shows by means of an example from the classical programming domain a more detailed description of the functionalities of these components.

Source code analyzer. Transforms the character string representation of the program into an internal representation in the form of symbol table and attributed parse tree. This transformation is based on the specific language whose specification is given. This program representation is then used as the basis for the program analyses to be done by most of the other tools.

Static analyzer. Generates most of the static information, program analysis, and structures such as flow graphs, call graphs, referenced-used information.

Dependency relation generator. Generates the dependency graph which is based on the dependency rules and the static representation of the program. This graph relates different program entities to those on which they depend, along with their dynamic conditions.

The dependency of different program entities on different attributes of other entities is based mainly on the specification of the programming language. Those dependencies may differ for different languages. Instead of implementing a different such tool for every language, the tool itself would take the language dependency rules as one of its inputs to build in the dependency structure which, once built, is independent of any specific language.

Consistency relation generators. Both static and dynamic consistency relation generators build the data structures necessary for constraint and consistency checks. They, based on the constraint rules, relate the program entities to the dependency and static analysis information. The main purpose of these operations is

to produce the necessary information to simplify and speed up the on-line checks.

Change request analyzers. These analyzers check the validity of change requests by applying all static and dynamic constraints before they are confirmed. Dynamic consistency checks must guarantee the consistency among the modified entities and the entities comprising the program execution history.

The static change request analyzer applies the static constraint rules defined and represented by the static consistency relation generator to the static program representation and the requested change. If the new program representation encompassing the change does not satisfy all the constraints, the change request will be refused and the user will be informed. A similar process holds for dynamic checks to guarantee history consistency.

Ripple effect analyzer. Accumulates the side effects of change requests which are then considered as new change requests. This analysis is based on the dependency relations relating different program entities to each other. Based on the rules describing the effects of the change (which are language dependent) the requested change will be studied and its effect on all related entities will be evaluated. Some changes have no side effects, while others have side effects on other entities which can be repaired automatically. Others have effects which can not be fixed without user interaction. Each of these effects is to be handled separately. If user interaction is required, a mechanism must be provided to inform the user. Effects which are automatically repairable are considered as subsequent change requests which must go through the whole process of validation and ripple effect analysis.

History update. This updates the execution history upon confirmation of changes. These updates are done by applying re-evaluation techniques based on dependency graphs. The history update operation works on a transaction basis. It starts its operation at the end of each change session. That is, when a change is requested and all its side and ripple effects are collected and have incrementally altered the static representations of the program, the change update starts its re-evaluation based on all the changes generated as a result of the originally requested change. The re-evaluation process assumes the existence of all the static updates in order to guarantee correct evaluations. All execution states must then be altered by confirmed changes starting at their point of effect specified as part of the change request. The re-evaluation process relies heavily on the dependence information produced by the dependence relation generator. Some of such dependences are conditional and do not apply to some execution states.

Incremental update. Adjusts all the program representations and analysis information according to the confirmed accumulation of changes in an incremental fashion.

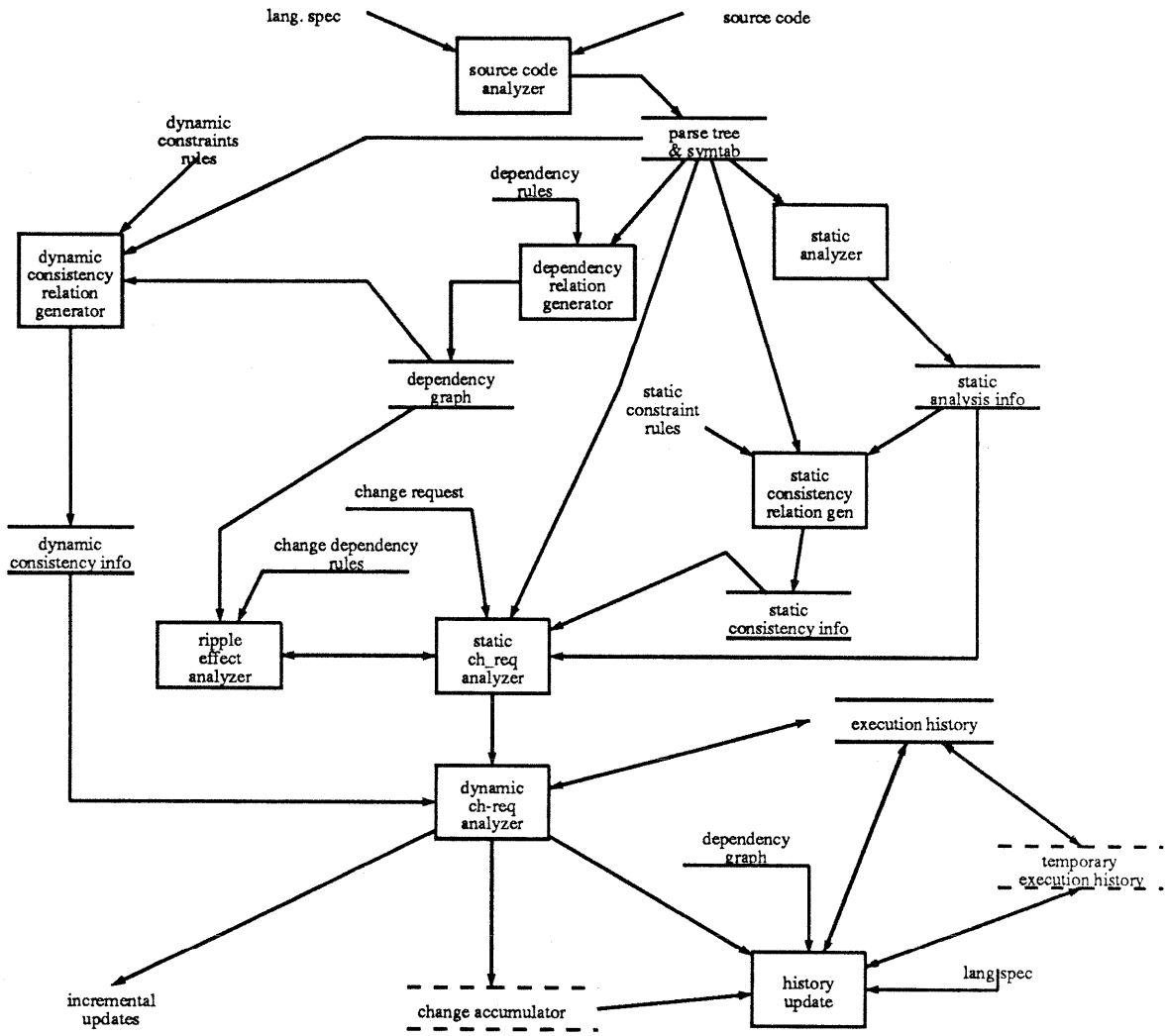


Figure 5
a. A Technical Maintenance Meta-Process Environment Tool Kit.

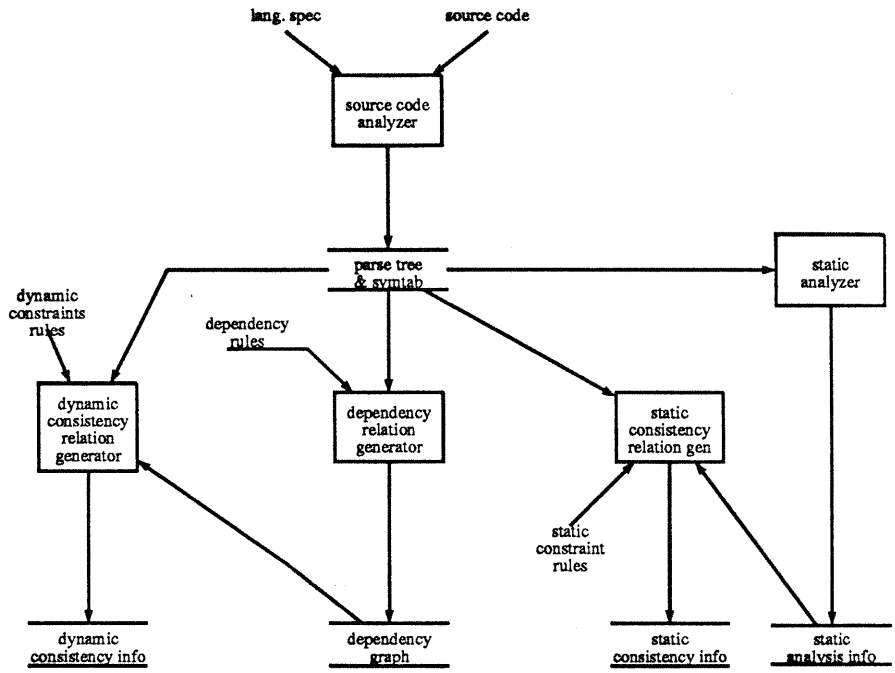


Figure 5
b. Program Static Representation and Analysis Tool Kit

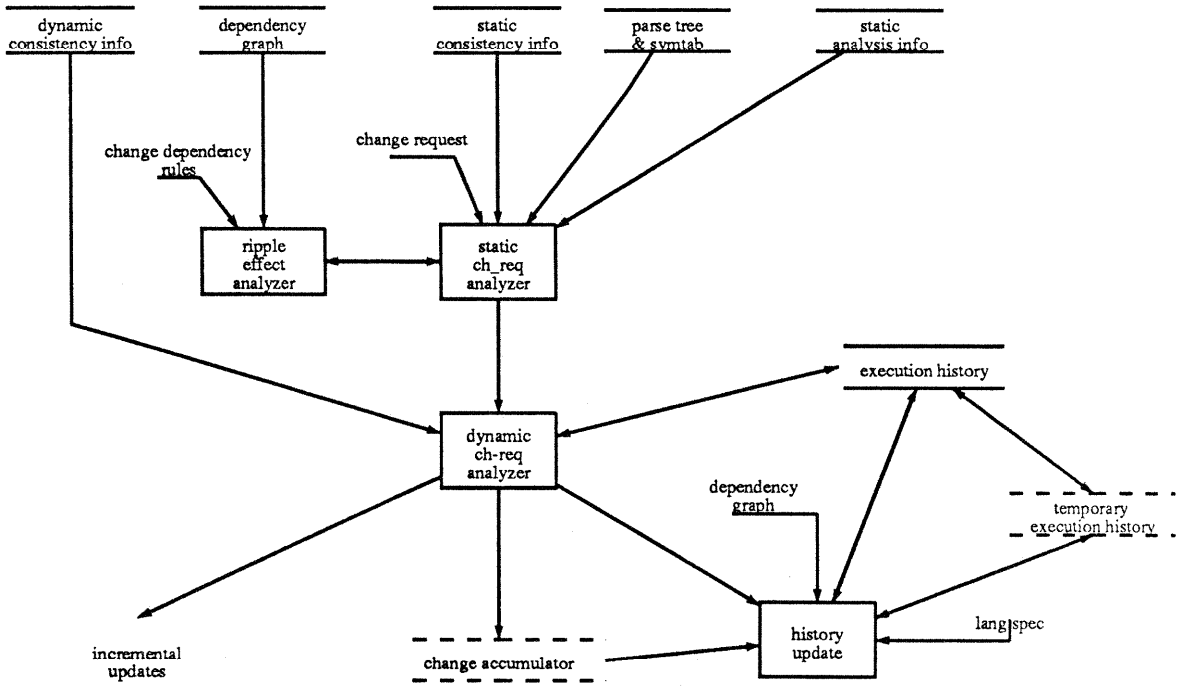


Figure 5
c. The Change Request Handling Tool Kit

5. Summary and Conclusions

Our research has convinced us that the process programming approach is very useful in helping to formalize and integrate the notion of software maintenance. Thinking of maintenance as a process seems to be a very natural intellectual activity, both for researchers and working software maintainers alike. Thus, it seems reasonable to use the process programming formalism to precisely characterize the differences between, and similarities among, the various approaches to maintenance.

We have begun to write process program fragments which express precisely what is meant by various types of maintenance, in order to better understand what is different about these various maintenance approaches, and in order to understand what common characteristics they have. We have concluded that maintenance is a software process whose purpose is to operate on an existing software product or process for the purpose of making changes in it. Various maintenance activities that have been described in the literature seem to be characterizable by differences in their process program descriptions. Differences in perspectives, for example between managers and practitioners, seem to be comfortably viewed as differences in the views projected by the running program. This seems to be analogous to the differences in the views that different debuggers might project to difference people during program execution.

Recognizing that software processes should be thought of as being software themselves has led to a number of important realizations and new insights in the maintenance context. First we now understand that maintenance processes must operate both on software application products and also on the software processes which are used to develop such products. The former type of maintenance is largely familiar to most software practitioners. The latter type of maintenance is far less widely recognized, as it is generally done informally by managers.

This informal software process maintenance is still more interesting because it is invariably done while the software process is still underway. Thus it is essentially a process of altering a system while it is still executing. We have denoted this type of maintenance, "dynamic maintenance." Consideration of dynamic maintenance then led us to understand that some types of dynamic maintenance are more complicated still, in that they require that histories of the progress of execution of the changed software process be altered to make them consistent with the new process. We refer to this as "history maintenance."

We have discovered that consideration of the nature, problems, and ramifications of dynamic and history maintenance is interesting, challenging, and of potentially great significance. Attempting to create a common framework for integrating our views of these new sorts of maintenance with the more classical forms has proved to be very valuable, resulting in a comprehensive framework within which we now believe we understand maintenance better, and which also suggests a closer kinship with development than we had previously expected to find.

We expect to continue to pursue this research in a number of ways. First, we will continue to develop process programs which we expect will lead to further elucidation of maintenance as an activity, and to better understandings of the different approaches to it. We hope to also gain a better understanding of the importance and wider applicability of dynamic and history maintenance. We believe that our understandings of these apparently new sorts of activities will have important implications beyond maintenance.

In developing more process programs to explore these ideas we will move in three separate directions. We will develop process programs to describe a wider range of maintenance activities, we will elaborate the process programs which we currently have to lower levels of detail, in order to make them more complete, and we will create more formalized specifications and designs of these process programs to augment our efforts to capture them in code. All of these activities will lead to better understandings of the differences and similarities in the domain of maintenance processes.

In addition, we are also beginning the development of a prototype maintenance environment. We will attempt to validate the ideas which we have expressed here by creating an environment in which various maintenance processes and subprocesses might be executed on actual software objects, such as code and requirements specifications. The architectural design of the environment has been completed and some details of the design have been developed as well. The environment will center on support for the creation of rather elaborate models of the relations among the various components of software objects. We have been impressed by how difficult it is to create a comprehensive model of these relations. More details of this model can be found in [.Gamalel-Din 88.].[5]

---

[5] To appear as a technical report very soon.

These models have been used as the basis for manual simulations of some low level maintenance subprocesses and we have been encouraged at how well they seem to support common and useful maintenance subactivities. We are now proceeding with our plans for implementing the maintenance environment and hope that as we proceed we will be able to validate our early ideas and gain new and deeper insights.

Acknowledgments

The ideas described here have been developed over a period of time. The authors have profited considerably from many useful conversations and discussions with a number of colleagues. Numerous discussions with Dennis Hiembigner, Bob Terwilliger, Stan Sutton, Mark Maybee, Xiping Song, and Fathy Eassa have all been quite useful in shaping these ideas. Conversations with Bob Balzer and Stu Feldman were also of great importance in improving our ideas.

In addition, the authors wish to acknowledge the financial support for this work which was provided by the Defense Advanced Research Projects Agency, through DARPA Order #6100, Program Code 7E20, and which was funded through grant #CCR-8705162 from the National Science Foundation. Additional funding for this work was provided through National Science Foundation grant #DCR-8403341.

References

[Agusa 85] K. Agusa, Y. Kishimoto and Y. Ohno, "A Supporting System for Software Maintenance", System Description Methodologies, editors: D.Teichroew and G.David, Elsevier Science Publishers, IFIP 1985.

[Arango 85] G. Arango, I. Baxter and P. Freeman, "Maintenance and Porting of Software by Design Recovery", Proceedings of the Conference on Software Maintenance 1985.

[Arcadia 87] R. Taylor et al, "Next Generation Environments: Principles, Problems, and Research Directions", Technical Report CU-CS-370-87, University of Colorado, Boulder, 1987.

[Balbine 82] G. Balbine, "Better Manpower Utilization Using Automatic Restructuring", in Techniques of Program and System Maintenance, editor: Parikh, Withrop Publishers, Inc, 1982

[Birrell 86] N. Birrell and M. Ould, "A Practical Handbook for Software Development", Camridge University Press, 1986.

[Bush 85] E. Bush, "The Automatic Restructuring of Cobol", Proceedings of the Conference on Software Maintenance 1985.

[Cashman 83] P. Cashman and A. Holt, "A Communication-Oriented Approach to Structuring the Software Maintenance Environment", in Tutorial on Software Maintenance, ed G. Prikh and N. Zvegintzov, IEEE Computer Society Press, 1983.

[Chikofsky 85] E. Chikofsky, "Application of an Information Systems Analysis and Development Tool to Software Maintenance", System Description Methodologies, editors: D. Teichroew and G. David, Elsevier Science Publishers, IFIP 1985.

[Clemn 84] G. Clemn, "ODIN - An extensible Software Environment Report and User's Reference Manual", Technical Report CU-CS-268-84, University of Colorado, Boulder, 1984.

[Dean 83] J. Dean and B. McCune, "An Informal Study of Software Maintenance Problems", Proceedings of the Software Maintenance Workshop 1983.

[Diaz 87] R. Diaz & P. Freeman, "Classifying Software for Reusability", IEEE Software, Vol. 4, No. 1, January 1987.

[Dowson 87] M. Dowson, "Iteration in the Software Process: Review of the 3ed International Software Process Workshop", Proceedings of the 9th International Conference on Software Engineering, 1987.

[Fay 85] S. Fay and D. Holmes, "Help! I Have to Update an Undocumented Program", Proceedings of the Conference on Software Maintenance 1985.

[Feldman 79] S. Feldman, "Make - a computer program for maintaining computer programs", Software Practice & Experience 9, 1979.

[Fosdick 76] L. Fosdick and L. Osterweil, "DAVE - A Validation Error Detection and Documentation System for FORTRAN Programs", Software Practice & Experience No. 6, 1976.

[Gamalel-din 88] S. Gamalel-din, "A Plausible Software Maintenance Process Program", Technical Report CU-CS-389-88, University of Colorado, Boulder, 1988.

[Gilb 82] T. Gilb, "Spare Parts Maintenance Strategy for Programs", in Techniques of Program and System Maintenance, editor: Parikh, Withrop Publishers, Inc, 1982

[Gilb 83] T. Gilb, "Design by Objectives: Maintainability", in Tutorial on Software Maintenance, editors: G. Prikh and N. Zvegintzov, IEEE Computer Society Press, 1983.

[Goldberg 83] A. Goldberg and D. Robson, "Smalltalk-80: the Language and Its Implementation", Addisson-Wesley series in Computer Science, 1983.

[Herndon 83] M. Herndon, J. McCall, "A Tool for Software Maintenance Management", Proceedings of the Softfair, Software Development: Tools, Techniques, and Alternatives, 1983.

[Higgins 86] D. Higgins, N. Zvegintzov, "Data Structured Software Maintenance: the Warnier/Orr Approach", Dorset House Publishing, co, 1986.

[Kaiser 87] G. Kaiser and D. Perry, "Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution", Proceedings of the Conference on Software Maintenance, 1987.

[Kishimoto 83] Z. Kishimoto, "Testing in Software Maintenance and Software Maintenance from the Testing Perspective", The proceedings of the Software Maintenance Workshop 1983.

[Letovsky 86] S. Letovsky and E. Soloway, "Delocalized Plans and Program Comprehension", IEEE Software, Vol. 3, No. 3, May 1986.

[Lientz 78] B. Lientz, E. Swanson and E. Tompkins, "Characteristics of Application Software Maintenance", CACM V. 21 No. 6, June 1978.

[Lientz 80] B. Lientz and E. Swanson, "Software Maintenance Management: a Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations", Addison-Wesley Publishing Company, 1980.

[Lientz 83] B. Lentz and E. Swanson, "Problems in Application Software Maintenance", in Tutorial on Software Maintenance, editors: G. Prikh and N. Zvegintzov, IEEE Computer Society Press, 1983.

[Martin 83] J. Martin and C. McClure, "Software Maintenance: the Problem and its Solutions", Prentice-Hall, Inc. , 1983.

[McClure 81] C. McClure, "Managing Software Development and Maintenance", Van Nostrand Reinhold Company, 1981.

[Narayanaswamy 87] K. Narayanaswamy and W. Scacchi, "Maintaining Configurations of Evolving Software Systems", IEEE Transactions on Software Engineering, V. SE-13 No. 3, March 1987.

[Neighbors 84] J. Neighbors, "The Draco Approach to Constructing Software from Reusable Components", IEEE Transactions on Software Engineering, V. SE-10 No. 5, September 1984.

[Osterweil 87] L. Osterweil, "Software Processes are Software Too", Proceedings of the 9th International Conference on Software Engineering, 1987.

[Parikh 82] G. Parikh, "How a COBOL Preprocessor Increases a Maintenance Programmer's Productivity.", in Techniques of Program and System Maintenance, editor: Parikh, Withrop Publishers, Inc, 1982

[Parikh 86] G. Parikh, "Handbook of Software Maintenance", John Wiley and Sons Publishers, 1986.

[Parnas 83] D. Parnas, "Designing Software for Ease of Extension and Contraction", in Tutorial on Software Maintenance, editors: G. Prikh and N. Zvegintzov, IEEE Computer Society Press, 1983.

[Peercy 81] D. Peercy, "A Software Maintainability Evaluation Methodology", IEEE Transactions on Software Engineering, V. SE-7 No. 4, July 1981.

[Perry 87] D. Perry and G. Kaiser, "Infuse: a Tool for Automatically Managing and Coordinating Source Changes in Large Systems", Proceedings of the ACM Fifteenth Annual Computer Science Conference, 1987.

[Pressman 82] R. Pressman, "Software Engineering: a Practitioner's Approach", McGraw-Hill Series in Software Engineering and Technology, McGraw-Hill Book Company, 1982.

[Rochkind 75] M. Rochkind, "The Source Code Control System", IEEE Transactions on Software Engineering, V. SE-1, December 1975.

[Schneidewind 87] N. Scheidewind, "The State of Software Maintenance", IEEE Transactions on Software Engineering, V. SE-13 No. 3, March 1987.

[Shneiderman 86] B. Shneiderman, et al., "Display Strategies for Program Browsing: Concepts and Experiment", IEEE Software, Vol. 3, No. 3, May 1986.

[Stankovic 85] J. Stankovic, "A Technique to Identify Implicit Information Associated with Modified Code", System Description Methodologies, editors: D. Teichroew and G. David, Elsevier Science Publishers, IFIP 1985.

[Stenning 87] V. Stenning, "On the Role of an Environment", Proceedings of the 9th International Conference on Software Engineering, 1987.

[Stroustrup 86] B. Stroustrup, "C++ Programming Language", Addison-Wesley Publishing Company, 1986.

[Swanson 76] E. Swanson, "The Dimensions of Maintenance", The Proceedings of the 2nd International Conference on Software Engineering, 1976.

[Tichy 85] W. Tichy, "RCS - A System for Version Control", Software Practice and Experience, V. 15 No. 7, 1985.

[Warnier 81] J. Warnier, "Logical Construction of Systems", Van Nostrand Reinhold Company, 1981.

Appendix 1

A Maintenance Meta-Process Program

1. A Process Program - 1.

The following is a maintenance meta-process program which considers both the static and dynamic views of maintenance. However, it only shows the part of the process concerned with handling change requests. The toolkit of the operators used in this program is shown in figure 5. Figure 1 also shows the relations among the different operators. It is clear that such a diagram shown in figure 5 entails no process description but is only a list of tools and data flow descriptions.

```
        --      global declaration of persistent objects.

Static_Analyzer    : Tool;
Static_Consistency_Gen : Tool;
Dependency_Graph_Gen_Tailor : Tool ;
Dynamic_Consistency_Gen     : Tool;
Ripple_Effect_Analyzer_Tailor : Tool;

process meta_maintenance1 (prog : Program, hist: Execution_History)

-- Declarations
--
  code_rep   is   handle for  record of
       parse_tree : Parse_Tree;
       sym_tab :   Symbol_Table;
  end_record ;

  lang_spec : handle  for  Lang_Internal_Representation ;
  static_analysis_info: handle for .....
       -- more detailed declaration is possible as shown above.
  static_consist_graphs : handle;
  static_constraint_rules : handle;
  dependency_graph : handle;
  dependency_rules : handle;
  dynamic_consist_graphs : handle ;
  dynamic_constraint_rules : handle ;
  change_dependency_rules : handle;

  change_request, chng : Change_Request ;
  change_req_pool, chng_acc : set of Change_Request ;

  hist_temp : Execution_History ;

  Dependency_Graph_Gen : Tool;
  Ripple_Effect_Analyzer : Tool;
  ..... -- other locally generated tools

-- Initializations Section:
-- To initialize and define all the
-- necessary information, e.g. the rules and specifications used
-- in the process program such as lang_spec as shown below.
-- Other rules that are assumed to be defined here are
-- constraint_rules, dependency_rules, ... etc.
--

 lang_spec := define_lang_spec ()
       -- this definition usually already exists in a library
```

```
        -- if not it could be evaluated through
        -- high interaction with the language specialist.

 if (not Object_mngr.does_exist(lang_spec) ) then
        Object_mngr.store(lang_spec) ;
        -- Storing the language specification is a process
        -- design decision, so it is kept outside
        -- "define_lang_spec" process.

-- Other specifications and rules are similarly defined and initialized here.

--      End initialization section and start of algorithm.


 Source_Code_Analyzer := Source_Code_Analyzer_Tailor(lang_spec) ;
        -- The Tailor works like compiler-compilers to
        -- generate a customized tool based on a given set of rules.

 code_rep  := Source_Code_Analyzer (prog);
        -- This information is not necessarily persistent.

 in parallel do
    {
        static_analysis_info := Static_Analyzer (code_rep) ;
        static_consist_graphs :=
                Static_Consistency_Gen (static_analysis_info,
                        rules=          static_constraint_rules);
                -- The explicit semantics are passed as parameters.
                -- Suppose that this tool is found in a non-tailorable
                -- form, it is still integrable to the process.
    }     ||
    {
        Dependency_Graph_Gen := Dependency_Graph_Gen_Tailor (dependency_rules);
        dependency_graph := Dependency_Graph_Gen(code_rep);

        dynamic_consist_graphs :=
                Dynamic_Consistency_Gen (dependency_graph, code_rep,
                        rules=          dynamic_constraint_rules);
    }
 end_parallel ;

 Ripple_Effect_Analyzer :=
        Ripple_Effect_Analyzer_Tailor(change_dependency_rules);

 Update_History := Update_History_Tailor(lang_spec);

 do until done
  wait_until change_request != NULL;
        -- This assumes that wait_until is a lnaguage construct. If not
        -- it may be implemented by using a interrupt mechanism
        -- or a read statement to read the requested change.

  change_req_pool += change_request;                -- set union
  hist_temp := Back_up(hist) ;          -- temporary history.
  while (change_req_pool != NULL)  do
   in parallel do
      {
        change_req_pool +=
             Ripple_Effect_Analyzer (change_req_pool,
                                dependency_graph,
                                static_analysis_info) ;
      }     ||
      {
       chng := Select(change_req_pool);
             -- Select is an algorithm to chose
```

```
            -- an element to work with.
            -- Making this selection done outside any of
            -- the tools shows the great customizability
            -- of process programs.
        change_req_pool -= chng ;
        if(not Static_Change_Analyzer(static_analysis_info,
                                      static_consist_graph,
                                      chng) )
        then ERROR;     -- raise exception;
        else
        if(not Dynamic_Change_Analyzer(static_analysis_info,
                                      dynamic_consist_graph,
                                      chng) )
        then ERROR;     -- raise exception;
        else
            incremental_updates (chng_acc)
                -- this is a tailorable process which contains
                -- different tools each for updating one set
                -- of information.
            chng_acc += chng ;
        end if;
       } ;
    end_parallel ;

    hist_temp := Update_History(hist_temp, chng_acc, dependency_graph);
    hist := Back_up(hist_temp);

  end while ;
 end do ;
end process.
```

2. A Process Program - 2.

The process program presented below describes another version of the maintenance meta-process presented in process program 1. Both programs uses the same operators and toolkit differently. The two programs are designed to show how different processes could be differently tailored to work under the same environment and toolkit. They also show that an integrated collection of tools is not enough to compose an environment.

The main difference between the two versions is that in process 2 the change updates are all directly applied to the persistent copies of the execution history and the program representations. This complicates the undo operation required when an error is encountered and a change is refused. This difference leads to the need for a persistent store of all program representations and tailored tools. On the other hand, it reduces the overhead of persistent data preparation and tool tailoring.

```
        --
-- Global definitions
--
-- As a design decision all the tools may be considered persistent --
-- The tailoring tools as well as all other tools including those
-- which are generated as described in this process program.
  code_rep   is   handle for  record of
       parse_tree : Parse_Tree;
       sym_tab :   Symbol_Table;
  end_record ;

  lang_spec : handle  for  Lang_Internal_Representation ;
  static_analysis_info: handle for .....
       -- more detailed declaration is possible as shown above.
  static_consist_graphs : handle;
  static_constraint_rules : handle;
  dependency_graph : handle;
  dependency_rules : handle;
  dynamic_consist_graphs : handle ;
  dynamic_constraint_rules : handle ;
  change_dependency_rules : handle;

  Update_History_Tailor : Tool;
  Update_History  : Tool ;
  ..... -- all other tools.

process meta_maintenance2 (prog : Program, hist: Execution_History)

-- Declarations, similar to that in process program 1.
--

  change_request, chng : Change_Request ;
  change_req_pool, chng_acc : set of Change_Request ;

  change_stack : Stack;  -- added to be used by the undo operation.

  hist_temp : Execution_History ;

-- Initializations Section:
--              Similar to that in process program 1.

lang_spec_name := get(name);
 Object_mngr.store(lang_spec) ;

-- Other specifications and rules are similarly defined and initialized here.

 Update_History := Update_History_Tailor(lang_spec);
```

```
Source_Code_Analyzer := Source_Code_Analyzer_Tailor(lang_spec) ;

Dependency_Graph_Gen := Dependency_Graph_Gen_Tailor (dependency_rules);

Ripple_Effect_Analyzer :=
        Ripple_Effect_Analyzer_Tailor(change_dependency_rules);

Update_History := Update_History_Tailor(lang_spec);

--    End initialization section and start of algorithm.


code_rep  := Source_Code_Analyzer (prog);
    -- This information is here made persistent along with all
    -- program representations.
    -- The scoping of these data are made global
    -- to this process program to reflect that they are persistent.

in parallel do
  {
    static_analysis_info := Static_Analyzer (code_rep) ;
    Object_mngr.store(static_analysis_info) ;

    static_consist_graphs :=
        Static_Consistency_Gen (static_analysis_info,
                rules=      static_constraint_rules);
    Object_mngr.store(static_consist_graphs);
  }   ||
  {
    dependency_graph := Dependency_Graph_Gen(code_rep);
    Object_mngr.store(dependency_graph) ;

    dynamic_consist_graphs :=
        Dynamic_Consistency_Gen (dependency_graph, code_rep,
                rules=      dynamic_constraint_rules);
   Object_mngr.store(dynamic_consist_graphs) ;
  }
end_parallel ;

-- initiate the dynamic maintenance part
do until done
  wait until change_request != NULL;
  dynamic_maintenance (change_request, hist);
end process;


-- The dynamic maintenance process

    dynamic_maintenance (change_request: Change_Request,
                hist: Execution_History)

-- A whole set of needed declarations as given in meta_maintenance2.

  change_req_pool += change_request;          -- set union

  -- No temporary history storage is employed.

  while (change_req_pool != NULL)  do
   in parallel do
     {
       change_req_pool +=
          Ripple_Effect_Analyzer (change_req_pool,
                    dependency_graph,
                    static_analysis_info) ;
     }   ||
```

```
    {
     chng := Select(change_req_pool);
          -- Select is an algorithm to chose
          -- an element to work with.
          -- Making this selection outside any of
          -- the tools shows the great customizability
          -- of process programs.
     change_req_pool -= chng ;

     change_stack.push (chng) ;
               -- the changes are stacked for later undo.

     if(not Static_Change_Analyzer(static_analysis_info,
                              static_consist_graph,
                              chng) )
     then
          -- undo the changes done so far.
          while not (change_stack.Is_empty) do
               chng := reverse(change_stack.pop);
                    -- The reverse operation is a complicated
                    -- one which must consider the reverse
                    -- actions in undo a change.
                    -- Undo the history update is not
                    -- directly reversible.
                    -- A complicated algorithm is required to
                    -- guarantee correct history reset.
               dynamic_maint(chng, hist)
               -- a recursive call of itself.
          end_while;
          ERROR;     --  raise exception;
     else
     if(not Dynamic_Change_Analyzer(static_analysis_info,
                              dynamic_consist_graph,
                              chng) )
     then
          -- undo the changes done so far.
          while (change_stack.Is_empty) do
               chng := reverse(change_stack.pop);
               dynamic_maint(chng, hist)
          end_while;
          ERROR;     --  raise exception;
     else
          hist := Update_History(hist, chng,
                              dependency_graph);
          chng_acc += chng ;
     end if;
    } ;
  end_parallel ;

     incremental_updates (chng_acc)
 end while ;
end do ;
end process.
```

Appendix 2

Dependency Relations as Basis for A Maintenance Environment

An example from the classical programming domain may help to explain the concepts of dependencies as well as dynamic maintenance and history update. An explanation of how the major components of the proposed maintenance environment described in section 4 may support the maintenance process will follow.

A classical binary search program.

A classical binary search procedure written in "C" programming language is shown below. It searches a sorted array "Tbl" of real numbers of length "nitem" looking for a value equals "item". If the value exists in the array it returns its relative location, otherwise a "NOT_EXIST" message is returned instead. The program source code augmented with line numbers is shown below.

```
--
0.    float  Tbl [UPPER_LIMIT];
/* Initialize the values of Tbl array */

1.  int       find_item (item, nitem)
2.    float         item;
3.    int           nitem;

{

4.    int           low,
5.                  hi,
6.                  mid;
7.    float         x ;


8.    low = 0 ;
9.    hi = nitem + 1 ;
10.   mid = (low + hi ) / 2 ;

11.   x =  Tbl[mid] ;

12.   while ((low < hi) && (x != item))
      {
13.         if (item > x)
14.         low = mid ;
            else
15.         hi = mid ;
16.         mid = (low + hi) / 2;
17.         x = Tbl[mid] ;
      } ;

18.   if (x == item)
19.         return ( mid ) ;
      else
20.         return (NOT_EXIST) ;
}   /* end find_item */
```

Dependency relations.

For the sake of simplicity, let us consider only one dependency relation type (object property) among the different program entities -- "value" dependency. The following notations are used in describing the dependency relations.

-->	:	depends on.
v(line-no)	:	variable "v" at line number "line-no".
E(line-no)	:	the expression at line number "line-no".
!	:	such that -- used to specify conditional dependencies. conditional dependencies are used for dynamic history update.
EX(line-no)	:	execution count of line number "line-no"
op(operations)	:	operators "operations".

The following is the set of dependency relations based on the above assumptions and described by the above notations.

1. low(8) --> E(8) ; E(8) --> const(0)

2. hi(9) --> E(9) ; E(9) --> nitem(1) & const(1) & op(+)

3. mid(10) --> E(10) ; E(10) --> hi(9) & low(8) & const(2) & op(+,/)

4. x(11) --> E(11) ; E(11) --> Tbl(..) & mid(10)

5. E(12) --> (low(8)!(EX(14)=0) or low(14)!(EX(14)<>0)) &
   (hi(9)!(EX(15)=0) or hi(15)!(EX(15)<>0)) &
   (x(11)!(EX(17)= 0) or x(17)!(EX(17)<>0)) &
   item & op(< , && , !=)

6. E(13) --> E(12) & item & op(>) &
   (x(11)!(EX(17)=0) or x(17)!(EX(17)<>0)) &

7. low(14) --> E(14) & E(13) & E(12) ;
   E(14) --> (mid(10)!(EX(16)=0) or mid(16)!(EX(16)<>0)) &

8. hi(15)--> E(15) & E(13) & E(12) ;
   E(15) --> (mid(10)!(EX(16)=0) or mid(16)!(EX(16)<>0)) &

9. mid(16) --> E(16) & E(12) ;
   E(16) --> (low(8)!(EX(14)=0) or low(14)!(EX(14)<>0)) &
   (hi(9)!(EX(15)=0) or hi(15)!(EX(15)<>0)) &
   op(+, /)

10. x(17) --> E(17) & E(12) ;
    E(17) --> Tbl & (mid(10)!(EX(16)=0) or mid(16)!(EX(16)<>0))

11. E(18) --> item & op(==) & (x(11)!(EX(17)= 0) or x(17)!(EX(17)<>0))

12. E(19) --> E(18) & mid(10)!(EX(16)=0) or mid(16)!(EX(16)<>0))

13. E(20) --> E(18) & const (NOT_EXIST)

The dependency description shown above is very useful not only for history update and dynamic maintenance but also for program understanding as well as assessing a change. The readability of such information may be improved if it is depicted by means of a graph.

To explain, let us study the dependency of a variable like mid(16) on different program entities. "mid" at line 16 depends on the value of the expression in the assignment statement in line 16. This expression depends in its evaluation on the values of "low" and "hi" as well as the operators constituting the expression. It also depends on the boolean expression of the while statement which controls the execution of the whole while loop. The values of both "hi" and "low", in turn, have their own dependencies which, eventually, depend on "mid" of line 16. This circularity is a direct consequence of the loop. It indicates that an expression depends in its evaluation on itself and hence help assessing the indirect effect of a change of an expression not only on different program entities but also on itself. However, it must be noted that such dependencies are conditioned, and hence dynamic dependencies are acyclic. Other dependencies may be understood by following the different relations described above.

Other dependency relations must also be considered, e.g. "variable type" dependency relation. Every variable is related to its type declaration which when changed, affects not only the variable value but also the operators involved (assuming the operator semantic's overloaded.)

Change requests Assessment.

Let us study the side and ripple effects of a change request from two different views -- The depth of ripple effect (the number of affected statements) and the method used for handling a change (local, automatic, or needs human interaction). Only static maintenance will be considered now. Dynamic maintenance will be considered shortly.

In studying the depth of the ripple effect, let us consider a change requested in the expressions at lines 10, 16, and 19. The change in the expression at line 10 will affect the statements at lines 10 - 19, which means that the whole program must be restudied. The change in the expression at line 16 has no effect on the statements of lines 10 and 11, however it has an effect on all other statements affected by the former change. That is its depth is 2 statements less. The third change has no ripple effect as it only affects the statement at line 19. The depth of the ripple effect of a change may be considered as a measure of the complexity of fixing the program based on the requested change.

Let us now consider the following three cases. The effect of changing the value of the constant "NOT_EXIST" is local to the point of change, i.e. line 20. It does not affect any other program point. Another example of a local change is changing the initialization of the array "Tbl". This kind of change has no static effect on the function "find_item". However, changing the type of "Tbl" from float to int, for example, needs user interaction in order to fix the type of "x" and "item", otherwise the comparison "!=" at line 12 will always be satisfied. As an example of a change from the third class, let us consider the change in a local variable's type, e.g. "low". Such a change has an effect on different program points specially from the compilation point of view, however they all could be handled automatically. This change has an effect on the operators' types but no effect on the overall program computation. This last conclusion could be reached by applying the change rules which are based on the language specification -- "a real value assigned to an integer yields an integer value".

Consistency checks.

As a quick example of a static constraint, let us consider the rule "if the control expression of a "while" statement is compound, the "while" construct must be followed by a conditional whose control expression must contain any of the "while" subexpressions or its negation". Now, if the expression at line 18 is changed, a violation of the above rule will be detectable.

Dynamic history update.

In updating the execution history of a program, only affected states and affected statements are considered. For example, let us consider a change in E(15), this change starts its effect on the program history only when statement 15 is first executed. Studying the dependency relations described above shows that all other program points will be affected which means that program execution may resume at the first affected

state. On the other hand, a change in the value of, say Tbl[5], in such a way that does not violate the dynamic constraint rule "Tbl must be sorted" will necessitate very few re-evaluations. This change starts to cause effects on the state when "mid" has a value 5. This effect is actually local to this state (in our example because Tbl is sorted.) Few re-evaluations (for statements 16, 12, and 13) are needed to lead to the conclusion that it it is safe to resume the original execution path with no effect on future states. This example suggests the plausibility of the re-evaluation technique.