ORBIT--A Prototype Software Maintenance/Development Process Programming Meta-Environment

Shehab A. Gamalel-Din, Leon J. Osterweil

CU-CS-428-89 October 1989

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

Orbit — A Prototype Software Maintenance/Development Process Programming Meta-Environment

Shehab A. Gamalel-Din

University of Colorado at Boulder
Department of Computer Science
Boulder, Colorado 80309

Leon J. Osterweil

University of California at Irvine
Department of Information and Computer Science
Irvine, California 92717

In this paper we introduce Meteor, an executable software process modeling formalism, which is designed around the idea of a process-centered software lifecycle paradigm. We also introduce Orbit, the meta-environment that supports the development of Meteor models.

Every environment should incorporate at least an implicit and preferably explicit model of the process it supports, and hence no single fixed environment can be expected to satisfy all users' needs. Users need to be able to adapt their processes and hence tailor their environments. Meteor is a process modeling formalism which captures most of the capabilities needed for modeling both static and dynamic views of software processes. Meteor is not only a process modeling facility but also an environment integration mechanism and a virtual machine for executing process models. The interconnection model supported by Meteor not only suggests a new model of software development but also proposes a facility for propagating and automatically manipulating maintenance requests applied to the model. Meteor components are reusable, programmable, and self adaptable.

The Orbit meta-environment prototype is designed to support software environment development by modeling of underlying development processes, using the Meteor formalism. Orbit applies maintenance techniques in developing process environments.

Orbit — A Prototype Software Maintenance/Development Process Programming Meta-Environment

Shehab A. Gamalel-Din

University of Colorado at Boulder
Department of Computer Science
Boulder, Colorado 80309

Leon J. Osterweil

University of California at Irvine
Department of Information and Computer Science
Irvine, California 92717

1. Introduction

Providing adequate support for developing and maintaining high quality software is a critical need. While environments to effectively support software development are now emerging, they provide only limited and incidental support for maintenance. Most people believe that "improving development improves maintenance," and that, therefore, software development environments also support maintenance. We believe that environments to support maintenance have additional and different needs. Because maintenance varies so widely in conceptualization and practice, it seems unlikely that any single environment (and certainly not one designed to support development) can satisfy all of these needs [Gamal 88-b].

A maintenance environment must be unusually flexible and extensible in order to meet the requirements imposed by different development processes. Maintainers need to be able to customize their environments by tailoring them and integrating new tools, and they also need facilities to support incremental adaptation of the environment itself to meet the continuous evolution of external conditions. We believe that allowing users to tailor and continuously change their environments is equivalent to allowing for the modeling of the processes which are to be supported by that environment. Thus we conclude that an effective maintenance environment should be based upon the notion of process modeling.

Process modeling helps maintenance in at least two key ways. First, if maintenance process models are sufficiently tangible and precise, they support process monitoring and evaluation that leads to adaptation and improvement. Second, explicit development models can be used to help capture development process information which materially facilitates the understanding and evaluation phases of maintenance [Fay 85].

Process programming [Osterweil 86, 87, Gamal 88-c] is a process modeling technique that is based on the notion that software processes should be considered to be software themselves, and that they can be explicitly modeled and programmed in the same ways that software products can. This implies that process programs themselves need support environments for their development and execution. That is, we need environments to support developing, activating, evaluating, and maintaining environments. Environments of this sort have been called meta-environments, [Sorenson 88]. Meta-environments support the tailoring of an environment by adjusting its specifications, within certain bounds. The environment thereby produced falls within a certain class which is implicitly defined by the meta-environment designer. Process

programming suggests using programming language features to specify this class. Thus programming techniques are used to explicitly model the maintenance process to be supported by the environment. In this sense process programming is like conventional application programming. On the other hand, software processes entail very long term execution, require persistent object manipulation, involve careful specification of human-computer interactions, and must be dynamically customized and evolved. These are important characteristics in distinguishing process programs from applications programs. Our efforts to understand and effectively model these distinctions have led to new perspectives on maintenance as a process and to an improved model of the entire software lifecycle [Gamal 88-c] — namely, the process-centered software lifecycle paradigm — which unifies development and maintenance for both products and processes. Supporting such a paradigm in the form of a meta-process environment is the main theme of this paper.

Section 2 gives a brief description of some of the essential characteristics of known software environments. This discussion is aimed at focusing on the failures of known environments in meeting important requirements and on those features which we feel are needed for process programming environments. Section 3 of this paper discusses the benefits of the process-centered lifecycle paradigm and characterizes maintenance from the perspective of process programming. It also highlights some of the demanding characteristics of a process programming support environment. In section 4 we introduce Meteor, a process modeling facility which is the core component and virtual machine of the Orbit environment as well as its major integration facility. Meteor is a programmable information flow-based tool for constructing environment prototypes. It provides automatic propagation and interchange of information between the different interdependent environment model components. Section 5 introduces the high level architecture of Orbit, the proposed meta-process environment prototype. Orbit supports both development and maintenance of process environments throughout their process-centered lifecycle. It also supports development as an advanced maintenance process. Section 6 presents a summary and some conclusions.

2. Background and Related Work

Early software support environments have had numerous severe limitations. They have been costly to develop, they have been hard to modify, they have implicitly supported relatively fixed processes, they have focussed on restricted aspects of the software lifecycle, and few have addressed maintenance.

Perhaps the best known and most numerous of these past environments have been *Language-centered environments*, such as Interlisp [Teitelman 81], and Smalltalk [Goldberg 84]. In these systems both the environment and the application program are embedded in the same address space as a single monolithic system. This makes all the features of the environment available to the application as building blocks. These environments support coding very well, but are ill suited to support phases other than coding. Such environments support maintenance in different ways. They provide maintainers with program browsers to examine program structure and interactively determine the scope of a change, however the user is fully responsible for determining his/her browsing algorithm. Furthermore, incremental interpretation and dynamic linking allow incremental implementation. The portability of application programs developed in language-centered environments is very poor because the programs are highly dependent upon, and integrated to, their environments. In addition, they are language specific and impossible to extend to cover phases other than coding.

Structure-oriented programming environment generation systems (such as the Cornell program synthesizer [Teitelbaum 81], and Gandalf [Habermann 86]), are language independent and highly portable, yet still support only the coding process. The graphical representations of program structures which they provide are very useful in both design and maintenance. Manipulating these program structures seems much easier and less error prone than manipulating code, for maintainers who are rarely involved in program development. The process supported by structured-oriented environments is difficult to extend to support other lifecycle phases, however, and so is the accommodation of new tools.

Toolkit environments, such as Unix, on the other hand, use operating system facilities to glue tools into collections. The intent is to provide a language-independent environment that supports multiple languages with appropriate tools. Such environments allow a high degree of tailoring but provide little process-based management of the use of these tools. Furthermore, they have a very simple data model for tool interaction and persistent object storage. No structure or semantic information is recorded with the data which results in tools with few incremental processing capabilities. Thus, although maintenance tools may be designed and integrated into such toolkit environments, these environments do not themselves provide much active assistance to the tools for maintenance of large systems. The simple data model of tool interconnection eases the integration of new tools into the environment, but process modeling and tool integration are considered the user's responsibility. Process modeling (and hence process environments) entails the integration of processes which interact with each other in complex ways, and therefore require highly structured data models. These data models need more advanced data and object management than those found in toolkit environments.

Method-based and *CASE* environments (e.g. VDM-based environments [Terwilliger 87] and IDE [IDE]) compensate for the loose coupling of tools in toolkit environments by representing a particular process based on a specific fixed development method which is to be used by individual developers in phases such as requirements analysis, specification, and design as well as in product and project management. Successful method-based environments have an underlying theoretical model against which a particular process description can be verified. Processes modeling semiformal methods, such as SADT, CORE, PSL/PSA, and SREM, as well as more formal methods such as those focused on Petri nets, state machines, and VDM, all exist. None of these methods

support all program development phases.

Although *meta-environments* [Sorenson 88] have been created to allow tailoring method-based environments within a predefined specific framework, they are still tied to one or two phases of the development cycle. Meteor provides a more flexible method for tailoring processes which can then, moreover, be viewed from different perspectives. In fact, the Meteor formalism seems able to cover the capabilities of all the methods discussed above.

In summary, different classes of environments use different techniques to achieve different goals and benefits. Full integration is achieved in language-centered environments by integrating the application with its environment into a single monolithic system. Although high power and increased programming productivity are observed, poor portability is expected. Portability is improved in structure-oriented environments which deal more with program structures. This approach also leads to increased maintainability and modifiability, but is poorly suited to supporting integration of new tools. Toolkit environments provide better support for integrating new tools, but rely upon a very naive notion of data types and incorporate no process representation at all. Simple processes exist in CASE environments in the form of limited and inflexible methods which aid one phase of the software cycle only. Furthermore, CASE environments do not effectively integrate the specification of human activities with the specification of tool supported activities. In addition, none of these environments provides explicit representation of the underlying process, if it exists at all, nor allows tailoring or adapting the process or the environment itself.

Process programming environments offer the promise of being more flexible and providing a broader range of support for the software development and maintenance processes. They provide mechanisms for supporting not only the production phases but also process planning, scheduling, control, and resource management in a highly flexible and programmable way. Above all, they allow tailoring and adapting the explicitly represented underlying process, and hence tailoring the environment itself to achieve evolving goals and meet changing constraints.

3. Process Programming and the Process-Centered Software Lifecycle

We seek to build environments capable of effectively supporting software maintenance, by making these environments tailorable, flexible, and extensible. Our approach is to support the explicit representation of maintenance processes as process programs which are to be interpreted by our environment. These process programs are to be developed by process engineers applying maintenance techniques with the help of the environment itself, and are to be maintained (in order to yield the flexibility we desire) with the help of the environment also.

3.1. New Perspectives on Maintenance in the Context of Process Programming.

Clearly process-centered environments place new and demanding constraints and requirements upon environment builders [Gamal 88-b,c]. We now summarize some of these requirements and constraints.

Explicit process representation. A *process environment* must contain an explicit model describing the process that it supports, and must support interpretation (execution) of that model.

Incorporation of Humans. The environment must support the definition of the roles of humans in the process (e.g. process organization and personnel, skills and education, and the roles and methods of communication.)

Openness to augmentation by new object types and tools. "Tools" in the

process programming context are analogous to operators in classical programming languages. Both the input operands and the output results of the operators are considered to be instances of types which must be defined in the process program. Thus process programs must be expressed in a language which is type and operator extensible.

Persistent object manipulation. Objects must be stored and managed by an object manager which incorporates a typing system, authorization access control, a locking mechanism, and a storage system. The object manager must be able to update existing persistent objects in accordance with new process changes to guarantee consistency.

Software processes have some other characteristics which make their support an even more demanding challenge. For example, they execute over very long periods of time, and they incorporate humans as execution agents. As a consequence an executing software process incorporates a learning process, which implies the need for continuous evolution of the process, even during the course of its execution. Hence, a process development support environment must be;

Customizable. The degree of support and leadership specified in a 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.

User-tailorable. Every individual involved in executing the process must be given the facility to tailor his or her share within the higher level management-drawn framework.

Incrementally implementable. Understanding of any process is usually gained by experiencing it, making it particularly hard to completely specify the process beforehand. This suggests that the process program is best evolved dynamically, and implemented incrementally.

Dynamically adaptable. Effective evolution of a running process has to be done in such a way that it does not cause the needless waste of software objects which have already been developed and that minimizes human efforts by maintaining the process execution history as well.

These characteristics demand the development of software processes which incorporate advanced maintenance techniques. These new techniques can be applied to both process and product maintenance yielding more powerful and sophisticated environments. These techniques provide new perspectives on software maintenance activities and their support environments [Gamal 88-c] which are characterized as follows;

Process maintenance. A process may be maintained for the sake of its own improvement, or to improve product quality.

Product-related process maintenance. In most cases the changed (maintained) process is executed from scratch to produce new products which have important similarities to the products produced before maintenance, but which have distinct desired differences as well.

Static Maintenance. The static descriptions of both products and processes must be subject to maintenance.

Dynamic maintenance. Iterative and continuous improvements to processes are to be expected, even during the courses of their execution.

Execution history maintenance. Dynamic maintenance implies that existing persistent objects must be updated in accordance with process changes to guarantee their consistency with the changes and to take full advantage of earlier human efforts.

Product-related process history maintenance. Classically, 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, maintaining the history of the reactivated process by changing the values of some of the previously developed persistent objects it deals with, is a more appropriate and efficient method for maintaining products.

The newly introduced maintenance notions of process and dynamic maintenance as well as the related ideas of history maintenance and product-related process maintenance form the basis for the new software lifecycle paradigm shown in figure 1 — namely, the *process-centered paradigm* [Gamal 88-c] which combines both maintenance and development.

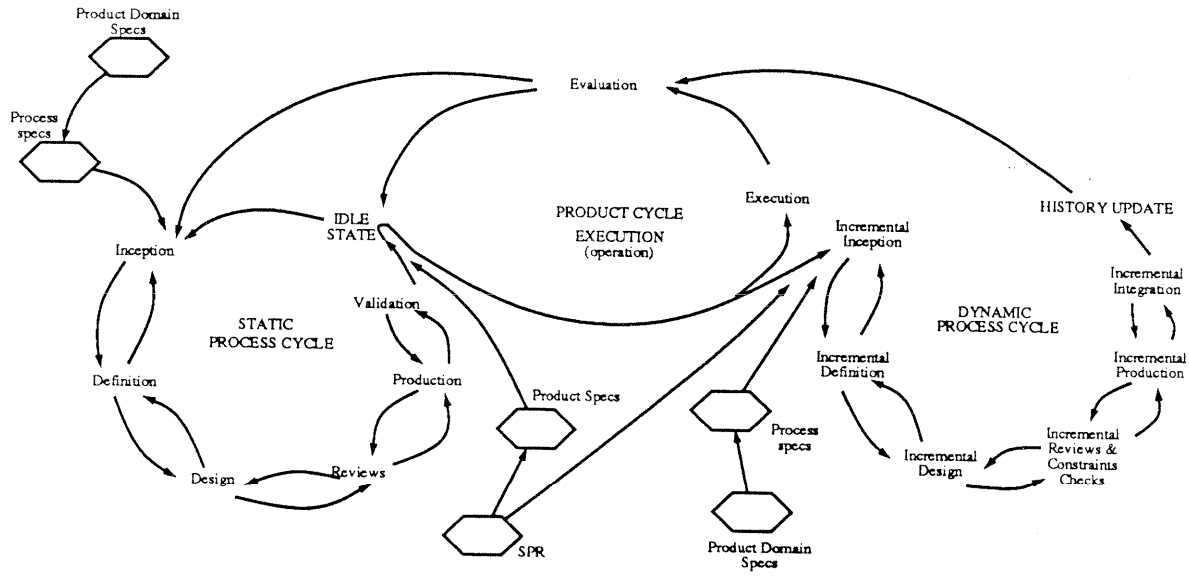

Figure 1. The Process-Centered Software Lifecycle.

3.2. Some Characteristics of a Maintenance-Based Process Environment.

Environments for supporting the maintenance of process programs will share some of the characteristics of environments for classical application 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 as automatic and powerful as possible, and it should support rapid responses to user queries. New sorts of analyses are required 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 (e.g., workers and managers) will each 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 a component of a software process operand (e.g. a requirement element) 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 reevaluate the program code involved in producing that component 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 operand type. Clearly it would be far easier to simply rerun the process program after each such modification, but this is usually impractical or impossible. Thus a process programming support environment should incorporate a facility for dynamic *reevaluation*. Reevaluation differs from reexecution 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 reevaluation 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 easy 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* (e.g., by means of assertions) and *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 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 [Feldman 79, Hudson 86, Clemm 89], 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.

To explain the differences between the two types of ripple effects analyses, let us consider, for example, the result of changing the value of an object at some point in the execution history. That may cause a ripple effect leading to the reevaluation of other variables and the eventual reevaluation of flow of control predicates which may then cause the execution of a different path through the program. Thus the environment must detect when reevaluation has caused execution of a new path, and must then roll back to the earliest deviation point, and restart execution from this point. It is even possible that a change may cause execution of a new path which criss-crosses the old execution path, raising the possibility of different rollback points. A specially devised algorithm may be needed to select an optimal roll back point from which to resume

execution. Looping causes even more serious problems.

Coarse grain module inter-relationship analysis is an excellent support for ripple effect detection in classical static maintenance. Finer granularity is required in the case of dynamic maintenance, however, especially for process programs, since this maintenance may be requested by users who are working at different metalevels and different levels of abstraction. Thus, what is considered to be an operator by some user, might be a process to other users. For example, process program maintenance may necessitate the replacement of a tool (operator) by another which may have either the same or different semantics. Portraying the impact of such a replacement to different users entails different analysis and different user interface capabilities. 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 led us to decide that even the basic elements of programming languages must be thought of as potential 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. Supplementary techniques such as history maintenance, execution monitoring, and dynamic constraint checking are needed as well.

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. 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 given language specifications for customizing generically built tools (e.g., 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. Figure 2 is a high level architecture describing the relationships among the different components comprising Orbit, the proposed prototype for a maintenance-based process environment. A more detailed description of Orbit's architecture is given in section 5, while Orbit's kernel and integration mechanism—Meteor—is introduced in the next section.

4. Meteor — A Process Modeling Formalism and Environment Integration Mechanism

Orbit views the process of modeling and developing software processes as an elaborate maintenance process. We believe that almost all the problems encountered in maintenance can be classified into two categories — problems related to program understanding, and problems related to studying the propagation and ripple effects of changes. Each of these types of problems is exacerbated by the unavailability in existing environments of explicit representation of the program constituents and their connecting and interdependence relations. Meteor supports the depiction of such relations in

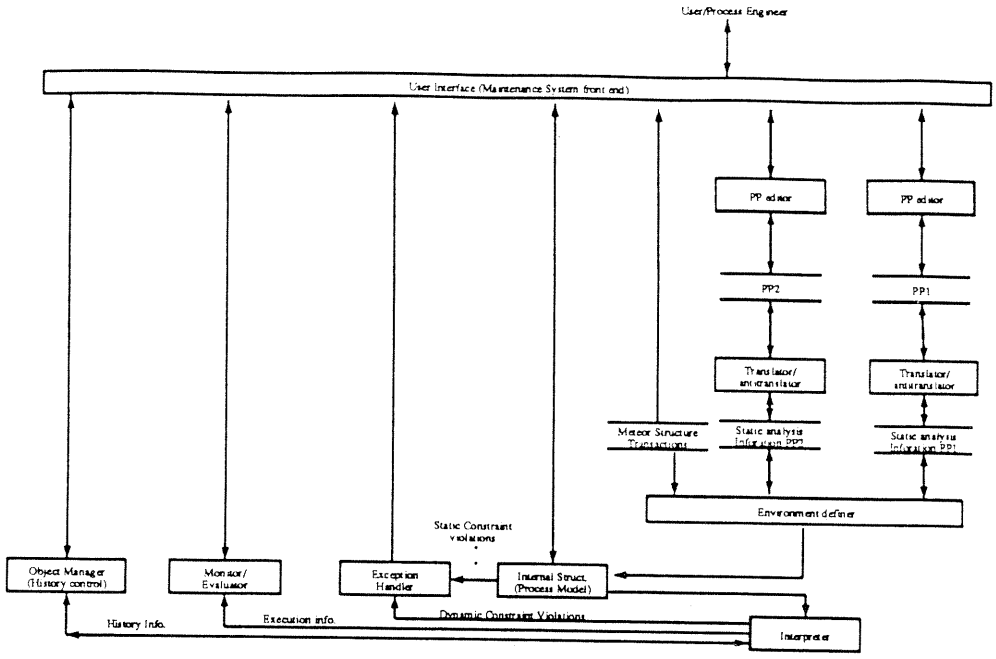

Figure 2. Orbit's Architecture for Executing and Monitoring Meteor Process Models.

different views and on different levels of abstraction in order to facilitate both static and dynamic maintenance of the model. Once the process model (program) is designed and its interdependence relations are established (hopefully from reusable process components), Meteor provides a mechanism for automating the propagation and analysis of changes applied to the model or its constituting entities. This is a considerable maintenance (and development) aid.

In fact, each of the various existing software process modeling methods is specialized to support a specific domain and view. Petri net-based models, for example, are used in the analysis of dynamic behaviors of concurrent processes [Peterson 81], while state machines (which in some cases are equivalent to Petri nets [Reisig 85] are good representations of program state evolution but they do not indicate how a transition is executed. On the other hand, CORE [Mullery 79] and SADT-based mechanisms [Ross 77] provide support for the static specification of environment components, but have a naive interconnection dependence representation model. Dataflow diagrams are another static modeling facility which completely lacks the notion of controlling the flow of data which we consider essential for process modeling. The interconnection and dependence information in VDM-based systems [Terwilliger 87] are totally implicit, and hence the complex task of understanding such relations is the user's responsibility. Process modeling needs powerful capabilities which allow viewing of the process from different perspectives and at different levels of abstraction. It also requires integrating both static and dynamic modeling techniques into a single and uniform technique. Meteor provides such a considerable and powerful modeling mechanism.

The brief introduction to Meteor given in this section is aimed at showing how Meteor provides a new paradigm for environment integration which is based upon explicit modeling of the underlying process. In these models, the different roles of the environment components are specified and instances of these specifications are interconnected in a simple manner to compose the program describing the process. User actions may cause automatic activation and execution of one of the processing cycles of the process, if these actions are execution requests. However, if user actions take the form of process maintenance requests, the model will automatically adjust itself to the changes requested and propagate the effect of these changes to maintain the previous execution history of the process.

Actually, Meteor is a programmable virtual machine for modeling and interpreting software processes. Process components (e.g. tools and humans' roles) communicate via flowing information carriers which capture properties of objects and store them in a powerful programmable common object base. It also controls the versions of objects produced by processes by maintaining the history of the properties of these objects. In fact, both the object base and the processes are all represented (simulated) in a common internal structure which explicitly represents most of the relations interconnecting process components and objects, and provides them with automatic maintenance facilities. Hence, there exist no actual bounds, at the interpretation level, separating any of the environment's components. In addition, both foreign and specially designed tools are treated equally in this system.

A brief description of Meteor, with a few elaborations where most needed, follows. A complete description of Meteor can be found in [Gamal 89].

4.1. Introduction to Meteor.

Meteor bases its modeling techniques on an adapted version of dataflow architectures [Thakkar 87] which, in eliminating unnecessary statement orderings, do a good job of showing the concurrency which is one key characteristic of software processes. Fortunately, dataflow diagrams had been proven [Kavi 87] to be equivalent to Petri nets [Peterson 81] for which there exists a large body of theoretical work [Reisig 85]. Thus we can draw upon this work in proving properties of the models built with Meteor.

Meteor models are composed mainly of two classes of components — objects and relations. Objects are interconnected by the appropriate relations. Relations are represented by frames which are programmable representations of the various relations' properties. All frames have input and output ports, and an object (or a frame instance) may be connected to any of the input or output ports of any other frame instance by means of a unidirectional pipeline. The direction of the pipeline determines the direction of flow of information carriers (tokens) from one node to the other in the model net, formed by the frame instances and connecting pipelines. Token flow is controlled by control constructs which conditionally synchronize and manage token routing and flow. Frame ports are typed so that when a composite token is received, an automatic coercion process takes place to filter out unwanted information. Ports are considered storage places whose ordering algorithms are predefined as part of their type definitions. They store tokens until their frames are ready to process those tokens.

An informal description and further refinment of Meteor's components are given in the next section. Figure 3 shows a simple example, the details of which are explained in Section 4.3. In this figure a Meteor model expressing the workings of a procedure "max" which computes the maximum of two values is shown. The "execution mode" of this model is programmed such that the "<" frame executes once it receives both tokens at its input ports. Those tokens, actually, carry all properties of the corresponding objects, e.g. value and type. In this example, the "<" frame is assumed to produce either a "Yes" or "No" token at its output port based on the result of the comparison. This

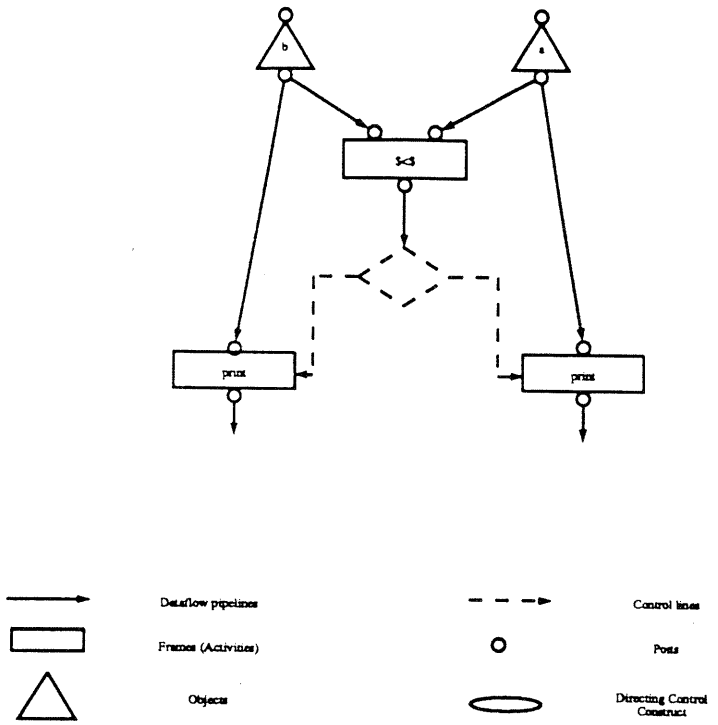

Figure 3. An Example of Modeling "max" with Meteor Formalism.

token is then used as the condition for the control gates which manage the flow of data to the "print" frames. To explain the model of information propagation, let us assume that each of the objects "a" and "b" has two attributes, namely value and type. Both of these attributes will flow through the whole net, however the control constructs will allow only one "print" frame to fire and execute. Since only the "print" frame needs value attributes, its input port will automatically detect and keep the attribute that is needed and filter the other one out. The importance of this model is not simply that it models the workings of the max function, but that it also models relations and dependencies which can be used to indicate all changes which must be made if certain changes to max (e.g. changing the types of its operands) are to be made.

4.2. Informal Description of Meteor Ingredients.

A frame contains a pre-programmed set of preconditions (firing rules) each of which is associated with an action which will be performed whenever this condition is satisfied. The frame fires by executing the actions associated with all the satisfied conditions in a predefined priority order. These conditions are defined in terms of the existence of specific information types at the frame input ports. The actions in the first prototype of Meteor are not restricted to a specific syntax or semantics, instead they are simple procedure/function calls. Frame firing may result in producing a new set of tokens at its output ports which will then be broadcast through the network of frames by means of the connecting pipelines and under the control of the control constructs in the network.

For example, when a token is produced at the output ports of object "a" of Figure 3, it automatically propagates through all the connecting pipelines to the input ports of both the "print" and "<" frames, according to the preprogrammed communication conventions of the pipeline. It should be noted that other modeling systems, such as SADT [Ross 77] and CORE [Mullery 79], provide similar capabilities for modeling systems by using frame like structures. However, their specifications and structures are not intented to execute, and hence the semantics of their models are purely static.

A Meteor object is a frame with an associated state. Every object has its own control methods to manage accessing its state as well as the state's history (versions.) An object's input port accepts tokens carrying property change information which is then used to update the object state and to generate appropriate output tokens to broadcast the change over the net. Thus every object is able to meet its own object management needs internally. Consequently, this approach does not mandate a single strictly specified object manager with limited and predefined capabilities, but rather allows the user to easily build and expand object management capabilities using the same method and structure which are to be used in modeling the process itself.

Tokens are used as information carriers to transfer information from one node to another in the model net. They take the form of a collection of attributes (the object's properties) with an indication of the recent actions (e.g. changed, deleted, etc.) to which the destinations are assumed to react. Because component types, such as frames, can be defined in isolation with no previous knowledge of how their instances are going to be connected, a token attribute representing a single object property may not always be understood at a destination port, and hence may automatically be filtered out. This actually matches the way maintainers usually think[1] and improves the model's understandability and modifiability. It also makes the component definitions highly reusable.

The communication model of Meteor is unlike other message passing systems as the communication partners have no predefined agreements on the message formats. It also differs from the object-oriented model of communication (e.g. in Smalltalk [Goldberg 83]) in which all objects are allowed to communicate with all other objects that understand their messages. These systems, theoretically, do not restrict communication to a limited set of objects and therefore allow malicious objects to be very disruptive. Maintaining and debugging such systems are not easy tasks. Pipelines are the major communication method in Meteor models. Although a pipeline is a unidirectional information flow channel, two-way communication can be provided by defining a pair of unidirectional pipelines. The method of communication carried out by those pipelines can be dynamically chosen from among several preprogrammed methods, e.g. remote procedure call, direct call, or token memory sharing. Such flexibility of communication enables the execution agents (e.g. frames) to be modeled on different processors of a pipelined machine or a network, or even by a separate active process on the same machine. This, in fact, matches our observations about models of software processes, and is consistent with our belief that humans should be considered to be execution processors.

Software processes need to *control* and *synchronize* the activation of their components. Every traditional programming language has its own paradigm of explicit or implicit control, and so should process programming languages. However, we believe that explicit control is more appropriate for modeling purposes, as this eases the model prototyping process and simplifies prototype evaluation and tuning. Meteor provides a primitive set of control constructs together with a set of combining operators which

[1] We believe that maintainers usually start by localizing their understandings and analyses before building a full model of their program. Delocalization is then gained by following interconnecting relations [Letovsky 86].

allows the user to expand and tailor the set of control constructs. Interestingly, this modeling approach draws heavily upon primitives used in logic circuit design. For example, conditioned gates control the flow of tokens in 1-M (multiplexors), M-1 (demultiplexors), or M-M (a combination of both) input-output gates. The conditions are logical expressions concerning token existence in specific places and ports. In addition, every component, frame, and control construct, has a condition port which plays an important role in deciding on the execution of a component. To explain, the tokens generated at the output port of object "a" of Figure 3 flows to both "<" and "print" frame input ports. However, although the firing condition of the "print" frame are satisfied, the frame will not execute until its control condition is satisfied, and this is determined by the "decision" component. These condition ports use a three-valued logic in which an undefined state is complementary to the Yes/No states. When a frame receives a Yes condition it starts checking its firing conditions. When it gets a No condition, it automatically disposes all the tokens available at its input ports. It keeps busy waiting until any of these conditions occurs.

Meteor also recognizes the need for *constraint propagation* and *consistency verification*. Object management systems need to maintain the consistency of their object bases in accordance with a predefined set of local constraints. On the other hand, the continuous evolution of process models, especially during the course of their execution, requires constraint verification after every transaction not only for the object base but also for the model structure. Every frame and object can specify its own set of constraints regarding its internal consistency and its connectivity. It may also impose some restrictions on the overall model structure. The former type of constraint is expressed in predicate calculus and is manipulated locally. However, the latter type is expressed in temporal logic and is handled globally by the underlying system interpreter.

Figure 4 shows the major phases of a Meteor model lifecycle—type definition, model (object manager and process program) construction, instantiation, and interpretation.

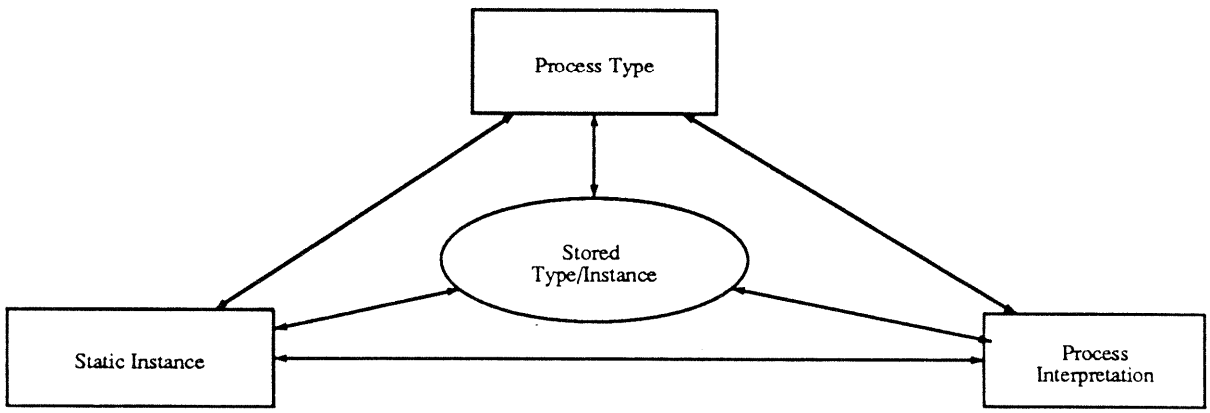

Figure 4. A Simplified lifecycle for Meteor Models.

Eventually, the process model including all its execution state history is stored in a process library. It then may be reactivated for product-related maintenance. This cycle may be interrupted at any time for maintenance purposes. A detailed description of each of these phases, together with the discussion of their roles in supporting process modeling according to our dynamic view of the process-centered lifecycle paradigm, can

be found in [Gamal 88-d].

4.3. An Example.

We now show how the Meteor formalism aids the maintenance process by automating change propagation and constraint verification. The example in Figure 3 shows a Meteor model of the "max" function. We now show how equivalent models can be automatically generated. If evaluation metrics exist, the best of these models can be automatically selected. This example will also show the benefits of Meteor for maintenance by showing how it supports the automatic propagation of change requests, and the concept of *dynamic maintenance*. Finally, the "max" model is used to demonstrate blackbox reuse [Gamal 88-c] by developing a model for "Max", the function that computes the maximum value of an list of objects.

Process Modeling. Figure 3 shows the representation of the semantics of "max" using predefined frames ("<" and "print") and control constructs. Frame types are expected to be defined from the specifications of the process/programming language used, while the definitions of control components are constructs of the Meteor formalism. Compound constructs, for both activities (frames) and control, can also be user defined. The semantics of the model can then be transformed into the equivalent pseudo code shown. Upon receiving input tokens representing the availability of the input parameters, the model executes as described in the previous section.

If the user decides to use a model of the same function, having a single exit, the model shown in Figure 5.a might be built. An optimization algorithm might even be used to eliminate both ":=" frames, and then generate the model shown in Figure 5.b which is equivalent to the conditional expression which might be written in C. Again, an automatic transformation can further refine the 2-1 gate into two 1-1 gates to result in the equivalent model shown in Figure 5.c which is almost identical to the one we started with, shown in Figure 3. Appropriate evaluation metrics can be employed for selecting the best of these models. Of course, several other equivalent models can be developed using different frame definitions.

Maintenance. Several kinds of maintenance and change requests can now be considered. Some of these requests may require changing some properties of objects, others may entail changing the model's structure. We now give some examples.

Assume that there are only two object properties for "a" and "b" in the "max" model shown in Figure 3, namely value and type. Now consider a maintenance request to change the value of object "b". For simplicity, let us assume a request to change the value in the current state rather than in any of the historical states. This change is represented by a request token which is imposed at the input port of "b"'s representation. Object "b" studies the effect of such a change on its properties and creates a new version of the changed properties. The result is then broadcast by the release of a token from its output port to the rest of the net. When this token reaches the input port of the "<" frame, that frame reacts. Note that the reaction of the frame in the context of this maintenance view should differ from its reaction in the normal execution view. In the change view the frame firing rule is programmed to require only one of its input ports to be enabled by a change request token, while in the execution view it will require both inputs to be enabled. When the port connected to "b" is enabled, the frame fires and pulls down the appropriate old value of "a" which is then compared with the new value of "b" and the result propagates. It should be noted that whenever a new value is found equivalent to an old value in a maintenance view of a model, token generation and propagation will stop. So if, for instance, the old value of "b" is found to be equal to the new one, the object "b" will not broadcast any tokens and no further propagation will take place.

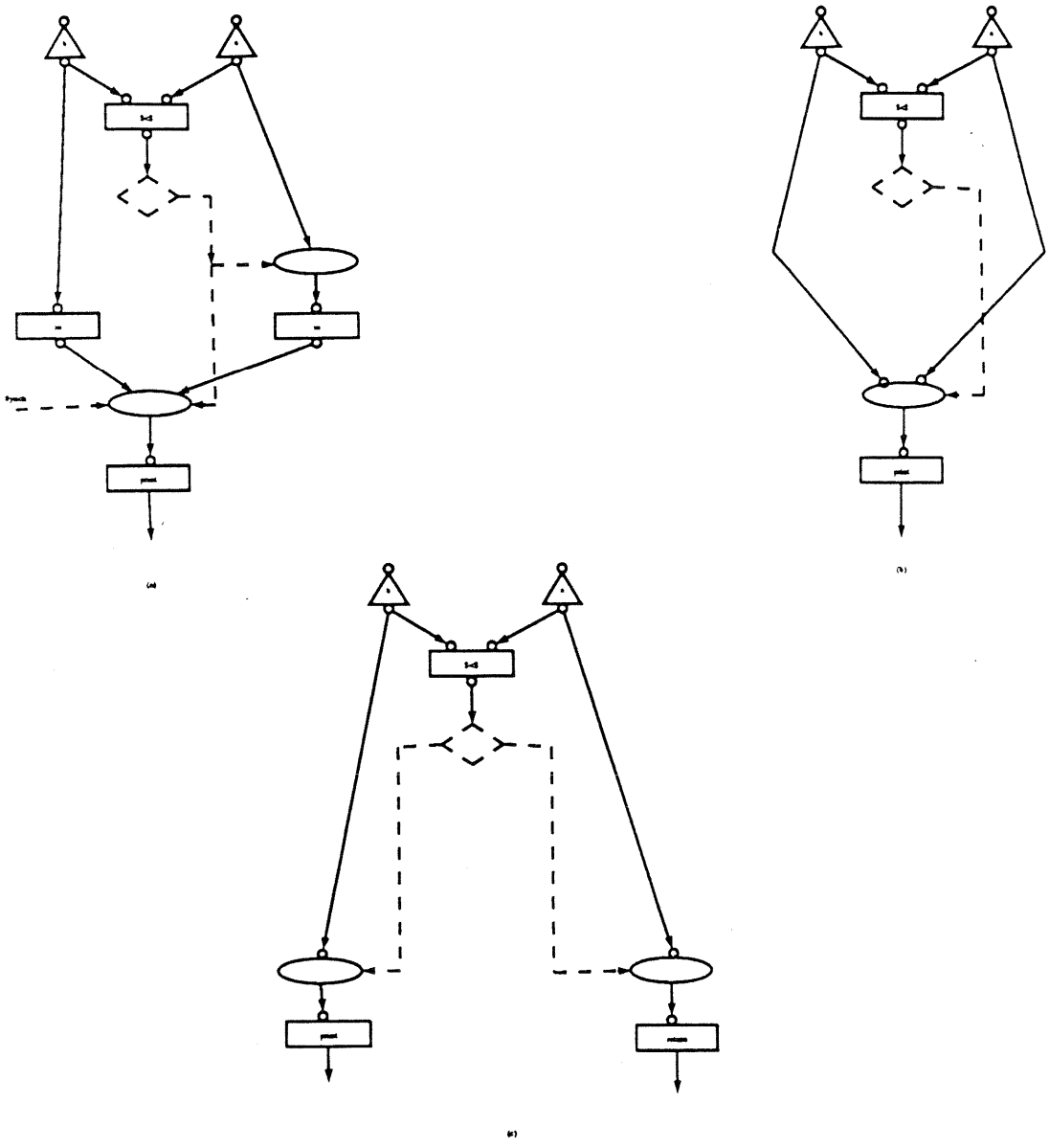

Figure 5. Different Models for "max".

Now consider a change request which asks for a change to the type of "b". In this case the value of "b" also may be affected and hence both of these properties will be broadcast. The "<" frame may be programmed to include type verification as part of its local constraints. The model will then behave as before if there are no violations. However if some constraints have now been violated, an error token must now be emitted. This may, for example, prompt the user and might cause the whole change request transaction to be reconsidered.

The program structure may be maintained by replacing, deleting, or adding any of the model components (including connecting pipelines.) Every component must be

programmed for the appropriate reaction to each of these kinds of change. For example, if the "<" frame in our example is replaced by a ">" frame to generate a "min" function model instead, the maintenance metalevel (see [Gamal 88-d]) to which both frame types belong will automatically take the appropriate action. This action is usually to release a set of tokens locally in the model, and then propagate them to update the execution history. So, in our example such a change will require pulling the appropriate values of both "a" and "b" down starting at the execution state at which the change is applied, and continuing execution as described earlier. Changes like disconnection, connection, addition, and deletion are treated similarly.

Figure 6 shows a model which demonstrates blackbox maintenance.

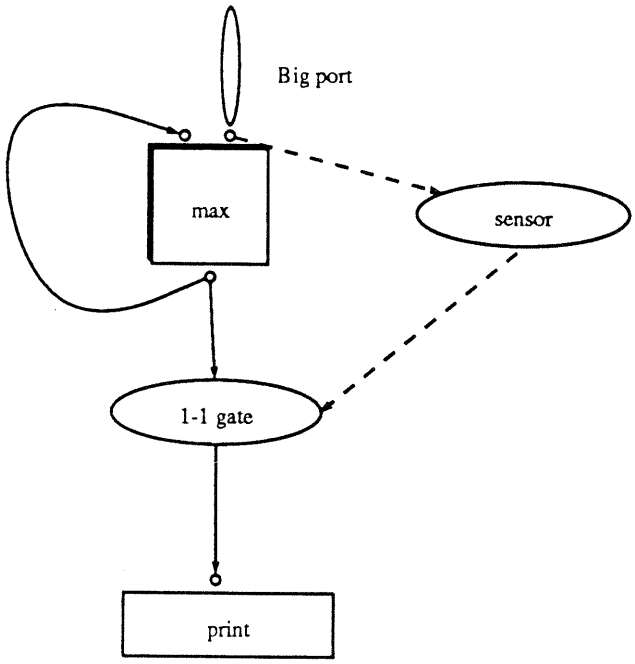

Figure 6. Meteor Model for "Max".

A model of a procedure which prints the maximum value of a list of objects is developed using "max". A new frame "Max" is defined which abstracts "max" into a single frame. "Max" has a queue type input port which will be connected to the list. "Max" will process one of the tokens in its input port at a time. No output will be printed except upon consuming all the tokens in the queue. It must be noted that in Meteor an undefined state is a real state which is represented by the nonexistence of tokens. So, "Max" can be programmed to print the value of the input token if the other port is in an undefined state.

This example indicates the use of Meteor as an environment integration mechanism. It shows not only how Meteor can integrate all tools and subprocesses through a common object base in which all objects are stored and managed, but also how it links all of these subprocesses in a uniform way for smooth information flow. Every subprocess may be abstracted into a single frame and connected to the other subprocesses.

Different abstraction levels as well as different model views are also supported. The independence of the component definitions and the information flow model of Meteor simplifies the development and integration of even foreign tools and subprocesses.

5. The Architecture of the Orbit Environment Prototype

The Meteor formalism has been found to be a powerful and flexible aid in modeling internal representations of classical application programs. It not only allows explicit pictorial representation of all implicit interdependencies and program relations but also supports the automation of change propagation. In fact, the Meteor representation provides the right support for process environments having such properties as dynamic and incremental adaptability, reusability of components, and new tool incorporation, as described in [Gamal 88-b,c]. Although Meteor provides the right mechanism for building such application environments, it itself needs a support environment.

Orbit is a prototype for a meta environment to support developing and maintaining process environments represented by the Meteor modeling formalism. It is designed around the environment requirements described in section 3. The Orbit design borrows features from earlier, more classical environments, to elegantly integrate applications and their environments into a single space allowing the user to manipulate the structure of both types of programs — process and application — without decreasing their portability. This is done by using the Meteor formalism.

5.1. High Level Architecture.

Before describing the environment structure and the semantics it provides for modeling processes, we briefly discuss the meta-process underlying the development of a new process environment and the interactions and roles of experts in this process. Figure 7 shows a process environment production process. The meta-process starts by defining the specifications of the application process based on available resources and requirements. These specifications are then compared against existing processes in a process library. The optimum process is then selected and statically maintained to fulfill the new requirements. An instance of this process can now be activated, and the product cycle starts (see Figure 1 for the process-centered lifecycle.) Tools, according to process programming, are considered as operators in the classical programming sense. New tools can be easily integrated into an application process after their semantics are described using the Meteor formalism. This formalism specifies the operational requirements of the tool as well as some maintenance-related knowledge (see section 8.) Continuous monitoring and evaluation can then be used to *dynamically adapt* the process to achieve optimum results. All of these activities are done incrementally so that they can be employed for both the static and the dynamic cycles of the process-centered lifecycle.

This process can start even with an only partially defined specification, such as a high level process outline. For example, Figure 8 shows how experts at different levels can interact with each others. When a higher level expert modifies the process at his or her level, other appropriate experts will then be prompted for *dynamic incremental adaptation* of their processes. This is typically how processes are developed through maintenance. The explicit definition of the roles, skills, and knowledge levels of experts is a step towards increased process automation. The application of expert knowledge seems effective in augmenting partially understood processes.

Orbit will also be able to readily support the *incorporation of new tools* into process environments, whether or not the tool was produced under Orbit-like support. Figure 9.a illustrates this. An example of a plausible structure of a two way translation system may be as shown in Figure 9.b. *Program portability* is supported because once such translators exist, a program developed outside Orbit may be easily integrated. Furthermore, this approach provides a highly *integrable multilingual environment*

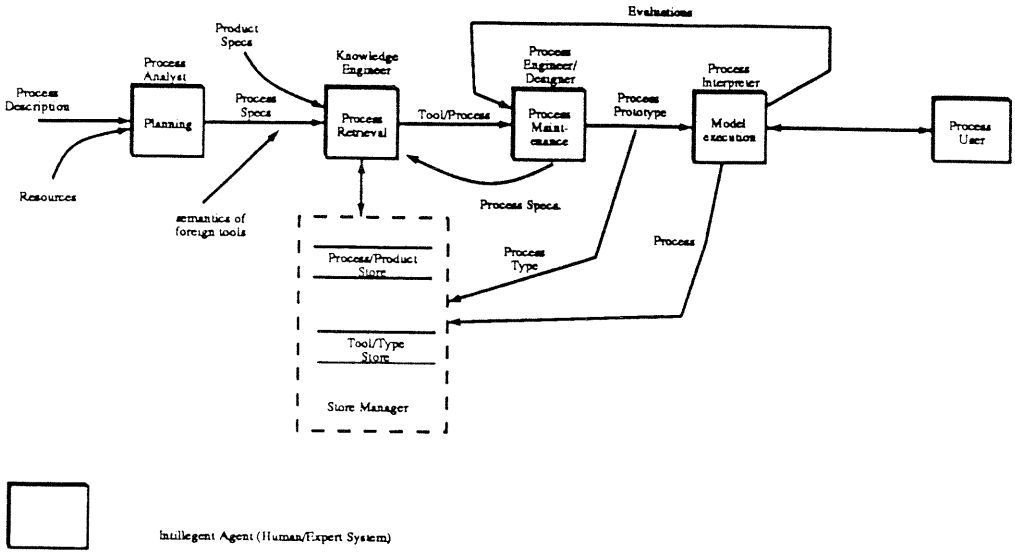

Figure 7. Abstract Model of Orbit's Underlying Process.

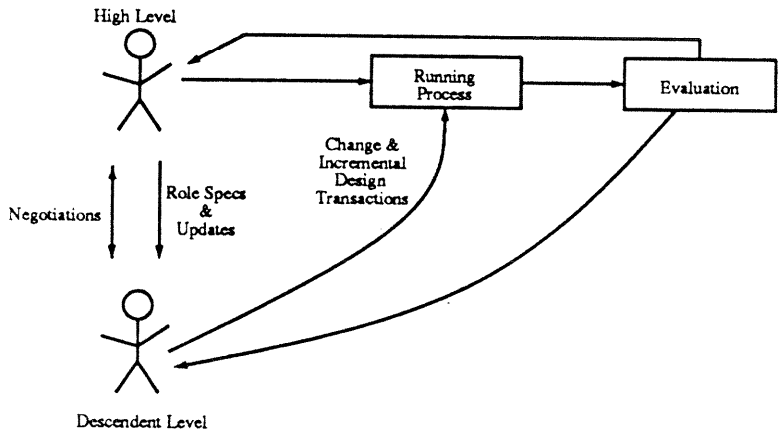

Figure 8. Orbit's Model of Incremental Dynamic Implementation and Adaptation.

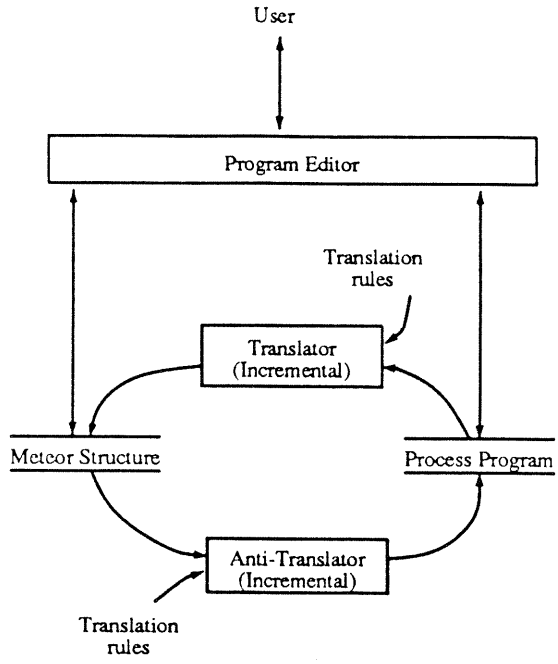

Figure 9.a. Two-way Translation for Meteor Models and Process Programs.

which easily incorporates foreign tools. It also generates highly *portable application environments.*

A process engineer or an expert may interact with Orbit in variety of ways. As shown in Figure 2 every process engineer can maintain his or her own view and abstract level of a process by maintaining either of its model representations — process program code in some textual representation or the unique representation of the Meteor internal structure formalism through a monitoring facility. Whenever an incorrect structure or inconsistent execution state is encountered due to either a faulty design or a change request, exceptions will be raised and appropriate parties will be informed. Maintenance requests are specified as nested transactions. At the end of every transaction or maintenance session, both the process model and its history of execution states are analyzed against their preprogrammed constraints. Temporal logic is used in Meteor to specify constraints on process execution histories. Appropriate constraint rules are automatically invoked whenever a relevant parameter is updated, and Orbit modifies state histories as needed. This *maintenance of execution histories* is an important and unique Orbit feature.

As noted earlier (e.g. see Figure 1 and Figure 7), *products can be maintained by maintaining their developing processes.* This implies that both the process and its products are considered to be integral components (see Figure 10) of Orbit. It also implies that once a process is instantiated and activated it never dies or terminates, unless that is explicitly requested (see Figure 11). Whenever a process description is needed to carry out a certain task, a new process instance will be created and activated. After

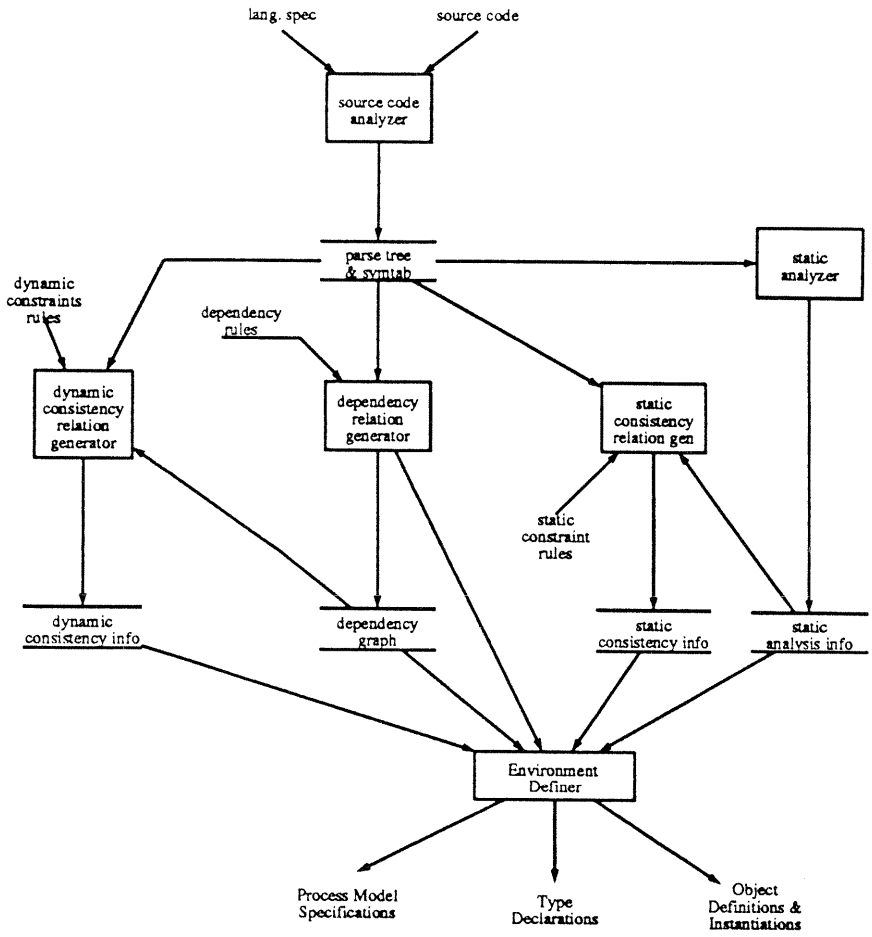

Figure 9.b. A Tool Kit Environment to Support the Translation Process.

finishing its programmed task, the process instance together with its products must be stored in a process store and kept in an idle state waiting for reactivation for product maintenance. Thus an execution state history is part of a process instance. This model unifies the definition of both classical programs and processes, in the sense that data objects never outlive their programs. Note that a process reactivation may lead to automatic reactivation of several other coprocesses which are all involved in supporting a higher level process. This is due to the interconnection of products and their processes as is shown in Figure 10. On the other hand, another instance of the same process program may also be instantiated for a different application or it may even be

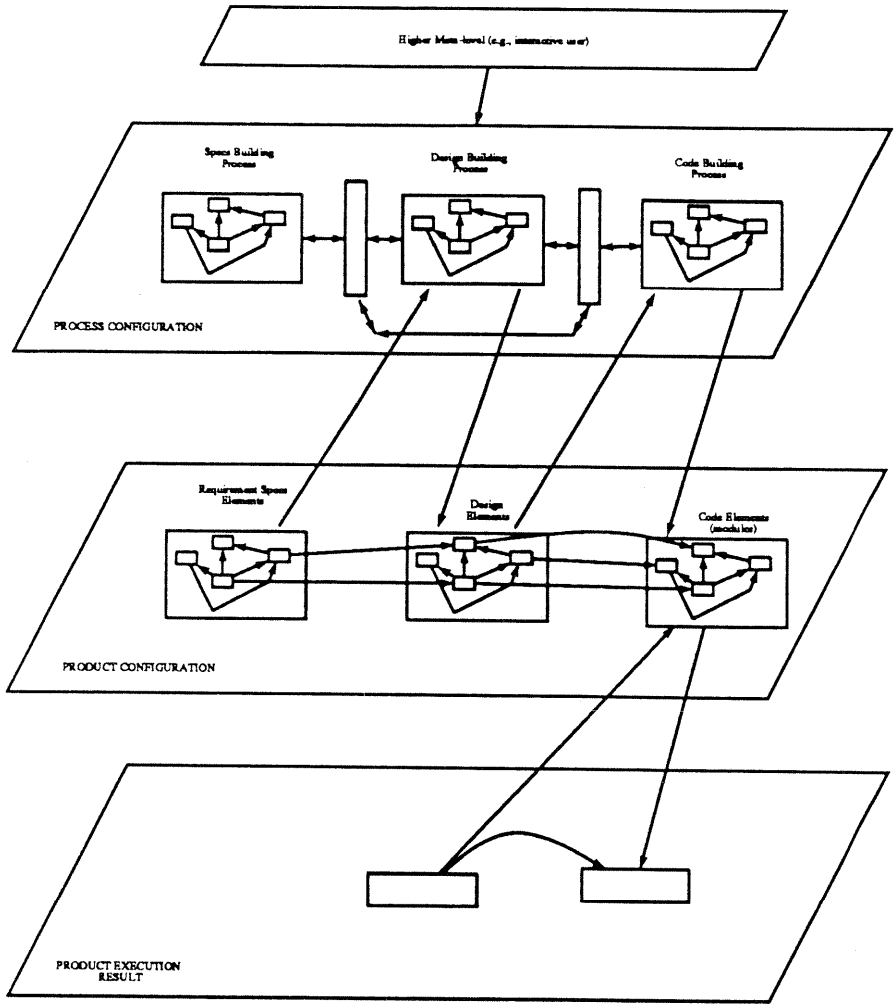

Figure 10. Process Models and Their Products are a Single Integral Entity.

statically maintained to produce a different version of the process description.

A simplified structure of Meteor, the central component of the meta-environment "Orbit" which supports process development/maintenance, is shown in Figure 12. Three main stores of information — a tool and process definition library, an object history, and a process model — are each incrementally managed by a separate manager, yet all are kept formally consistent. The Meteor formalism is used to represent stored information for each store. Any incremental environment modifications are directed to the appropriate maintainer (manager) of each of these stores for evaluation, checking, and integration. Modifications to any of these stores may automatically trigger other

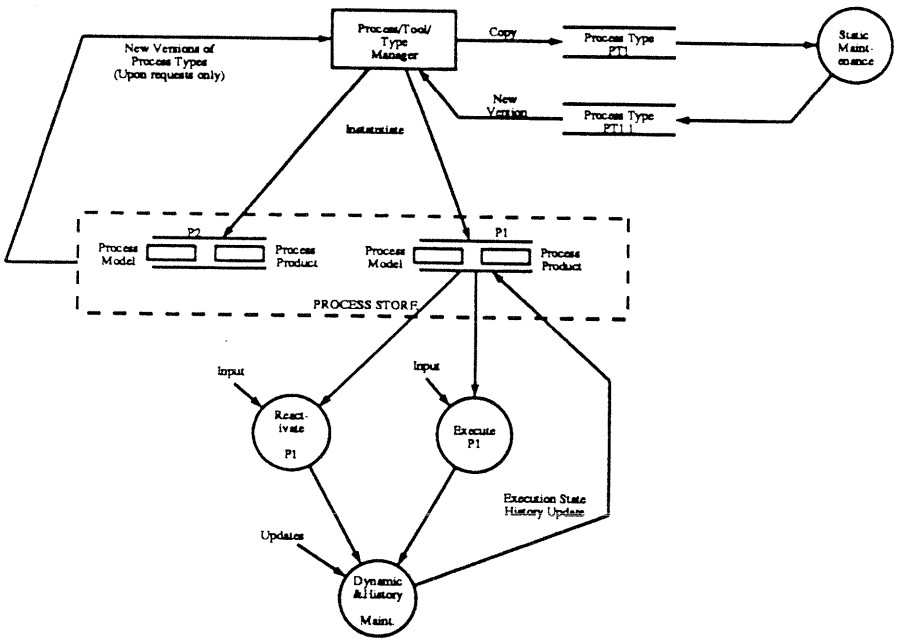

Figure 11. A Model for Dynamic Instantiation, Execution, Retrieval, and Maintenance of Meteor Models.

actions which may propagate to other stores depending on the interconnection of the process model components and their semantics. Detailed descriptions of each of these stores and how they define the objects with which they deal are omitted here, but can be found in [Gamal 88-d].

Finally, it is worth noting that we consider the user interface to be an important environment integration component. The user interface descriptions for every subprocess in a process environment are assumed to be defined by the tools operated by these subprocesses. However, it is important that the user must feel no differences when moving among the different subprocesses, because of differences in their user interfaces. Meteor supports this by using its internal representation formalism at the lower levels to represent the appropriate interfaces to various tools. A similar mechanism (e.g. Chiron [Young 88]) which is outside the scope of this paper must exist for higher level user interfaces.

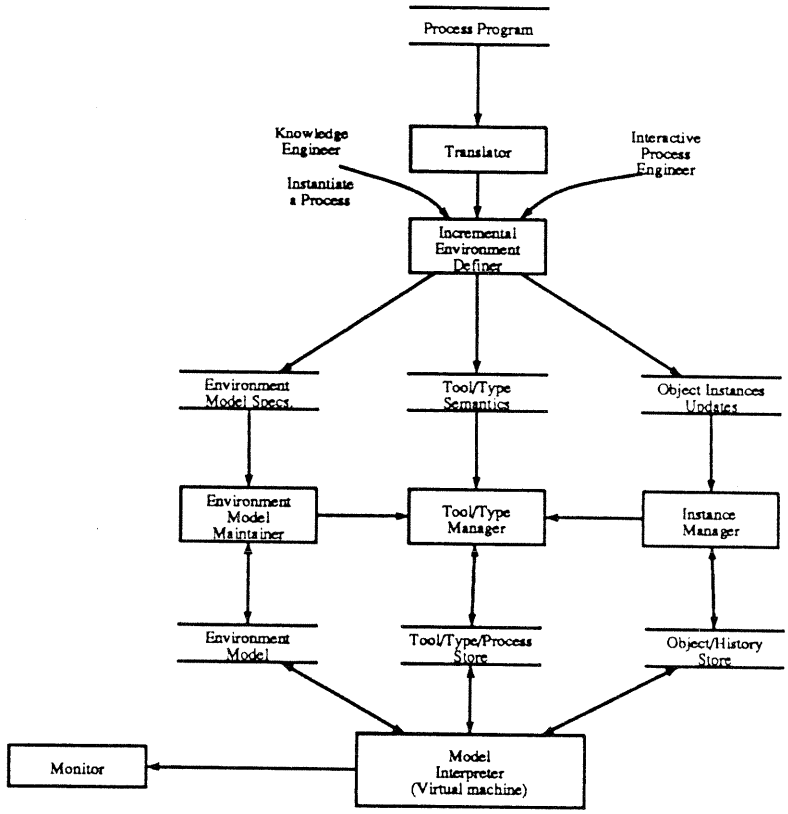

Figure 12. The Architecture of Meteor's Interpreter and Support.

6. Conclusions and Future Work

We will continue this research in a number of ways. Although we have modeled several programs in both the process and application domains using Meteor's formalisms, we expect to continue developing more complex models which will lead to further elucidation of software process modeling, and to better understanding of the different approaches to it. In developing more process models we will move in three separate directions. We will develop process models to describe a wider range of software processes, such as requirement specification, design, and coding processes. We will also elaborate the process models which we currently have to lower levels of detail in order

to make them more complete and to gain deeper understanding of how humans perform such tasks so that we can determine those parts of a process which can be automated by expert systems. These two directions will serve as a testbed for validating the ideas presented here. They will also help us to improve the Meteor formalism. Although Meteor is a plausible visual process programming formalism, we feel that we still need to create more formalized specifications and design a higher level language front end which will allow us to capture Meteor formalisms in code. So, we will also try to use these models to understand how users like to interact with the system and how they like to specify their processes so that we can develop a user-centered process programming language which captures the Meteor formalism.

Meteor has been developed in C++. In addition, we are also beginning the development of an Orbit prototype. We will attempt to validate the ideas which we presented here, especially those related to our new maintenance perspectives and to using advanced maintenance techniques in development, as product-related process maintenance is an essential goal of this research. We also want to investigate the power of Meteor as an environment integration facility and virtual machine. In addition, although the current implementation of Meteor is running on a single processor machine, the inherent concurrency support characteristic of Meteor encourages us to consider studying its implementation on an MIMD machine or computer network.

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

[Adler 88] M. Adler, "An Algebra for Data Flow Diagram Process Decomposition", IEEE Transactions on Software Engineering, Vol. SE-14, No. 2, February 1988.

[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.

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

[Clemm 89] G. Clemm and J. Osterweil, "A Mechanism for Environment Integration" to appear in TOPLAS, 1989.

[Concepcion 88] A. Concepcion and B. Zeigler, "DEVS Formalism: A Framework for Hierarchical Model Development", IEEE Transactions on Software Engineering, Vol. SE-14, No. 2, February 1988.

[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.

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

[Gamal 88-b] S. Gamalel-din and L. Osterweil, "Software Maintenance as a Programmable Process", Technical Report CU-CS-390-88, University of Colorado, Boulder, 1988.

[Gamal 88-c] S. Gamalel-Din and L. Osterweil, "New Perspectives on Software Maintenance Processes", Proceedings of Conference on Software Maintenance, 1988.

[Gamal 89] S. Gamalel-Din , Ph.D. Thesis Dissertation, University of Colorado 1989.

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

[Habermann 86] A. Habermann and D. Notkin, "Gandalf: Software Development Environments", IEEE Transactions on Software Engineering, SE-12 No. 12, December 1986.

[Hudson 86] S. Hudson and R. King, "CACTIS: a Database System for Specifying Functionality-Defined Data", Proceedings of the Workshop on Object-Oriented Databases, 1986.

[Kavi 87] K. Kavi, B. Buckles, and U. Narayan, "Isomorphisms Between Petri Nets and Dataflow Graphs", IEEE Transactions on Software Engineering, Vol. SE-13, No. 10, October 1987.

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

[Mullery 79] G. Mullery, "CORE - A Method for Controlled Requirement Specification", In Proceedings of the 4th International Conference on Software Engineering, 1979.

[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 86] L. Osterweil, "A Process-Object Oriented Centered View of Software Environment Architecture", Technical Report CU-CS-332-86, University of Colorado at Boulder, 1986.

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

[Peterson 81] J. Peterson, "Petri Net Theory and the Modeling of Systems", Prentice-Hall, Inc., 1981.

[Reisig 85] W. Reisig, "Petri Nets: an Introduction", Springer Verlag, 1985.

[Ross 76] D. Ross, "Structured Analysis (SA): A Language for Communicating Ideas", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977.

[Sorenson 88] P. Sorenson, et al, "The Metaview System for Many Specification Environments", IEEE Software, March 1988.

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

[Teitelbaum 81] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", CACM Vol. 14, No. 9, September 1981.

[Teitelman 81] W. Teitelman and L. Masinter, "The Interlisp Programming Environment", IEEE Computer, Vol. 14, No. 4, April 1981.

[Terwilliger 87] R. Terwilliger, "Encompass: An Environment for Incremental Software Development Using Executable Logic-Based Specifications", Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1987.

[Thakkar 87] S. Thakkar, "Dataflow and Reduction Architectures", IEEE (selected reprints), 1987.

[Waters 82] R. Waters, "The Programmer's Apprentice: Knowledge Based Program Editing", Transactions on Software Engineering, V. SE-8, No. 1, January 1982.

[Young 88] M. Young, R. Taylor, and D. Troup, "Software Environments Architectures and User INterface Facilities", Transactions on Software Engineering, V. SE-14, No. 6, June 1988.