

# Proscription versus Prescription in Process-Centered Environments

Dennis Heimburger  
University of Colorado  
Boulder, CO 80309-0430

## 1 The Psychology of Process-Centered Environments

Implicit in the concept of a process centered environment is the idea that a process program controls the interactions of programmers with the environment. The exact means by which this is to be accomplished is still a subject of some dispute.

The degree to which a process-centered environment is accepted will depend both on its ability to enforce a specified process and on its ability to support a non-restrictive style of interaction with programmers using that environment. Enforcing a specified process can be considered the *raison d'être* of process-driven environments. It would seem to be a requirement that an enterprise can define the process by which it wishes to construct software and be reasonably certain that programmers will adhere to that process.

On the other hand, human psychology dictates that programmers (they are human, after all) will not operate in an environment that is overly constraining. They need to perceive themselves as having some choice in their activities. There is a conflict then, between a desire to enforce a specified software process, and the need of programmers to control their activities. This tension seems an inherent problem for process-centered environments, and it is important for any such environment to come to some reasonable compromise around this issue.

I have chosen the terms “prescriptive” and “proscriptive” to describe this conflict. Obviously there is spectrum, and I have chosen to focus on the extreme points to illustrate the issue.

## 2 A Prescriptive Process Environment

I use the term “prescriptive” to indicate that the process environment closely controls the means by which a task is to be completed, and the order in which tasks are to be performed. I will refer to prescriptive environments and prescriptive processes interchangeably.

From the point of view of process programming, this implies a procedural, serial program that enumerates the tasks in a specific order and requires programmers to complete each assigned task before beginning another. In effect, prescriptive processes wish to treat “programmers as procedures” or “programmers as machines.” Stated so baldly, of course, no one would accept such a characterization as reasonable, but that is the effect of an extremely prescriptive environment.

A typical method of providing a task to a programmer involves a “task specific shell.” This shell provides a fixed set of tools by means of which the programmer is expected to accomplish the task. When the programmer is “in” that shell, only those tools are available for performing the steps of the task. Further, the process may place a great deal of structure on the usage of those tools, and require that a programmer perform task steps in a rigid manner.

This description of a prescriptive process/environment is intentionally extreme and it is only fair to point out that there is a great deal of merit in prescriptive approaches. For example, many interactive tools that programmers use, typically editors of some kind (textual or graphical), are quite prescriptive and are generally accepted. I would argue that this is because they are operating in a restricted domain around some particular data structure such as a piece of text, a requirements graph, or such.

Editors are a specific example showing a more general feature of prescriptive systems: they can provide substantial support for domain-specific activities because they of necessity incorporate a lot of knowledge about the activities being conducted and semantic knowledge about the tools being used and the ways they can be combined. Planning systems are good examples of the way that this can be exploited.

As a side benefit to domain specificity, prescriptive environments can provide better and more immediate feedback to the programmer when he or she deviates from the specified process.

One other major reason for using this approach is that it is obvious and easy. If one already has some form of process model that describes the steps to be taken for some process, it seems like a small and relatively easy step to interpret the model and make programmer tasks invoke a shell to control those tasks.

Finally, prescriptive processes have the advantage of being understandable. To the extent that programmers can understand procedural, serial, code, they have some chance of reading and understanding the process embodied in such code, and from the point of view of educating programmers, this is desirable. This also applies to the process in execution; its execution state is amenable to understanding and this in turn implies that programmer-managers can gain some insight in the progress of a process.

The primary demerit of a prescriptive system is that it is a bad match for programmer psychology. In its worst form, it can take away control, stifle creativity, limit programmer flexibility, and over-constrain task solutions.

One other, less important, problem involves process evolution. If some unforeseen situation occurs, then it seems plausible that temporary changes to the process will be needed to accommodate it. This seems to be a difficult task when the process program is itself rigid.

### **3 A Proscriptive Approach**

In contrast to the prescriptive approach, one can envision a “proscriptive” process. The term “proscriptive” indicates an environment which operates by prohibiting inappropriate programmer actions and otherwise does not constrain the means or order in which tasks are performed.

Such an environment can be characterized as result-oriented and constraint-based. That is, the process does not necessarily say *how* to solve a task, rather it says *what* the result should be from solving the task and leaves it to the programmer to construct such a solution. In this class of environment, programmers are free to use whatever tools they desire in order to complete a task. They then communicate to the task sufficient information (typically some data structure) and an indication that the task is completed.

Just because a programmer proposes a solution does not mean that it is acceptable to the process. The motto here is “trust, but verify.” The process cannot always trust the programmer’s solution. So, associated with tasks in a proscriptive environments is some set of constraints that specify what it means for a proposed solution to be acceptable. If the proffered solution is deemed inadequate by the task, it can reject the solution.

The merits of proscriptive processes tend to mirror the demerits of the prescriptive environment.

1. Provides a good match for programmer psychology. By definition, a proscriptive environment does not care how a task is solved, only that a solution is presented, so the environment is certainly not too restrictive.
2. Additionally, control no longer strictly resides with the process. Rather, it establishes more of a peer relationship between the process and the programmer. They communicate back and forth “negotiating” acceptable solutions to tasks.
3. Unforeseen problems should be easier to handle. Often what is required is to temporarily remove some constraints so that tasks can be solved in an order or by a means not originally contemplated.

By contrast proscription fails where prescription can succeed.

1. It is more difficult for a process to make use of domain-specific knowledge; it has little information about and no control over the activities of a programmer.
2. Feedback is delayed until the programmer provides a solution. This may be well after the point at which some error was committed, and may be difficult to provide a reasonable description of why some solution was unacceptable.
3. Understanding proscriptive processes is difficult. In some sense there is no process to see, only a lack of one. Understanding and monitoring such processes is difficult.

## 4 Unifying Proscription and Prescription

Originally, I had felt that the more that an environment matched the proscriptive paradigm, the better the environment. After much discussion inside the Arcadia group, and most notably with Lee Osterweil, I have become convinced that environments must support both. The less constraining proscriptive approach is essential if the environment is to be used by real people. The proscriptive approach is necessary in order to manage the process, to gain insight into its progress, and to allow programmers to understand what is the process.

Constructing unified processes is a matter of programming style, but it is also a matter of programming language features. It is difficult to follow a good style if the language does not support it with a reasonable notation.

In the course of examining process programming languages, it is possible to identify some characteristics that appear to support a unified programming style.

**Peer Communication and Parallelism:** It is important that neither the process nor the programmer always be in control. This seems to indicate the need for a more peer relationship. My interpretation of this is that the environment should be treated as a collection of communicating process programs, some of which serve as surrogates for programmers.

**Procedural Processes:** The prescriptive nature of a unified language can be handled by making the individual processes quite procedural. Thus it is possible to examine processes and understand and monitor them.

**Constraint Enforcement:** Some means for specifying and enforcing constraints is needed. One way is to define a specific constraint sub-language; this is the approach used in APPL/A, for example. A somewhat more primitive method, but ultimately more general is to provide for general event management. Examples are APPL/A triggers and SoftBench or FIELD broadcast messages.

**State Query:** In order to monitor the progress of a process (or processes) the state must be available and in a persistent form. The language must allow some means of querying that state.

## 5 Conclusion

A number of the languages that exist have the potential to provide a reasonable mix of proscription and prescription; Merlin, APPL/A, AP5 could all, I think, be made to work with some effort. I believe that the problem to date is that most efforts have been concentrated on trying to model processes at all. That is, process description is hard, and doing it with “style” has not been an issue. But it is now time to attempt this style of programming, probably by starting from existing, prescriptive process models and trying to add in proscription. The tradeoffs will be difficult to make, but the effort should be rewarding.