An Analysis of Process Languages

Stanley M. Sutton, Jr., Peri L. Tarr,
Leon J. Osterweil

CMPSCI Technical Report 95–78
August 1995

Laboratory for Advanced Software Engineering Research
Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This work was supported by the Advanced Research Projects Agency under grant F30602-94-C-0137.

Abstract

Process programming represents the application of the software engineering idiom and technology to the development of superior software processes. There is a substantial and growing body of research into software process technology in general and process programming approaches in particular. Much of this research concerns process languages. We are now in a position to draw some conclusions about the ways in which process languages may contribute to software process engineering. Furthermore, we can begin to make inferences regarding the original hypothesis that process programming is an applicable and feasible technology for software process support.

We take the fundamental aim of software process research to be the production of better quality software in less time and at less cost. Thus, we emphasize issues in linguistic support for software production processes. These issues include the distinction between process requirements and process language requirements, language requirements for the representation and execution of production processes, granularity of process representations, and process visualization. We also examine issues of language definition and use, specifically alternative architectures for linguistic support of processes, and issues related to meta-capabilities and meta-processes, especially as they affect production processes.

Analysis of these issues leads to the identification of particular programming approaches that are more or less able to contribute to particular problems in software process engineering. It further allows us to draw inferences about the fundamental applicability and feasibility of process programming to software process engineering. These inferences concern the importance of code and non-code representations, the diversity of language paradigms, the importance of rigorous semantics, and the need for and difficulty of process evolution.

1 Introduction

Over the past several years there has been a growing conviction that software systems should be viewed as products resulting from the execution of orderly software development processes. There is, further, a belief that software development times can be reduced, and software product quality can be improved through the disciplined application of superior software processes. These beliefs have served to focus attention on the problem of creating and exploiting superior software processes.

As an alternative to developing an entire new discipline of process development, it was suggested that process development might be viewed as a type of software development [29]. This suggestion seemed attractive as it suggested that much existing software technology might be exploited to support the development of processes. Thus it was suggested that software processes should be modelled with software modelling techniques, coded with coding languages, and evaluated and evolved with corresponding software technologies.[1] As this suggestion offered the hope of avoiding the creation ab initio of an entirely new discipline, it seemed eminently worth investigating through a program of research.

Since that time there has indeed been a great deal of software process technology research. Much of this research has centered around the development and evaluation of process technology that borrowed heavily from either the idiom or the technology of software development. Thus there have been numerous projects aimed at exploring the suitability of traditional approaches to modelling, coding, evaluating and evolving software as paradigms of how to do these same things for processes. While the evaluations of individual prototypes from these projects has yielded much insight into the various approaches, we believe that the community stands to gain even more from an evaluation of the original premise—namely, that process development can be effectively supported by adapting software development idioms and technologies. Specifically, we believe that, after nearly ten years of experimentation with software process programming and process programming languages, we can now draw some conclusions about the ways in which process languages may or may not contribute to the development of superior processes. In turn, this allows us to begin to evaluate our initial hypothesis, namely, the feasibility and applicability of the programming paradigm to software processes.

In this paper we identify some of the more common goals for software process support, especially as related to production of software products, and we evaluate how successfully

[1]It is our view that the term "process programming" should cover the entire range of software development activities, from requirements through evaluation and evolution. Thus we often use the term "programming" synonymously with "software development" but not synonymously with the narrow activity of "coding."

various process technologies have addressed these goals. In many cases we are able to identify some lessons that have been learned from not only one, but many, projects. It is a particular goal of this paper to focus our evaluations on what we have learned about how to adapt software development idioms to meet stated software process objectives. Often what we have discovered is that programming approaches are indeed useful and, further, that the weight of experience strongly indicates particular approaches and technologies that seem especially effective in addressing particular process problems.

The evaluations presented here should not be taken to be complete and comprehensive, as space does not permit this, and the body of experience is still growing. Instead, in this paper, we summarize some of the more salient and important lessons that seem to be inferable from significant bodies of research based on some of the more popular and important process research approaches. The paper concludes by attempting to draw broader inferences about the validity of the basic hypothesis that software development technology can form an effective basis for process technology. In some areas these inferences seem safely drawn. In other areas, useful inferences cannot safely be drawn due to a lack of sufficient prototyping and experimentation. A summary of these latter inferences then comprises a research agenda for the coming years.

2 Analysis of Issues

This section analyzes process development and language issues in three general categories: support for the software production process, our main emphasis (Sections 2.1 through 2.4), process language definition and use (Sections 2.5 and 2.6), and support for meta-processes (Sections 2.7 and 2.8).

2.1 Process Requirements versus Language Requirements

The goal of software processes is to facilitate the development of high-quality products more quickly and at lower cost. To address this goal, it has been proposed that software processes should be represented as software and subjected to rigorous software engineering technology. This technology should include specification, execution, analysis, testing, measurement and evaluation, and so on, both to improve both the process and the resulting product. Toward this end, suitable languages (to which we will refer as *process languages*) are required.

Software processes must accommodate human development activities. These are characterized by creativity, judgement, tentativeness, incompleteness, informality, and lack of structure, and by activities that are either partially or not at all supported by automated tools (e.g., meetings, negotiation, etc.) (see [2]). Contrast this with the goals of process languages, which are to facilitate the definition of well engineered, high-quality software processes to which the software engineering discipline can be applied. Such software processes must be characterized by formality, rigor, completeness, well-formedness, and semantic depth. The goals of software processes and the languages that support their definition and execution are clearly distinct, as are the respective sets of requirements they must satisfy.

Despite the differences between process goals and process-language goals, process-language researchers have tended to treat process requirements as process language requirements, with the end result that many process languages include support directly (in the form of language constructs) for features required of processes (e.g., [3, 10]). This often leads to the problem of proliferating language features ("kitchen sink" languages), which are difficult to use, analyze, and apply to multiple domains. It may also result in process languages that have characteristics that make them less well suited to the application of rigorous software engineering. Process languages are required to capture and support the essential characteristics of software processes, but not necessarily to mimic them. Process characteristics are not determined solely by language characteristics. Rather, process characteristics are determined by a combination of language characteristics, process model or program design, and run-time factors.

To demonstrate why separation of process and language requirements is desirable, consider the issue of support for process state reification, which entails the representation of process execution state in the form of meta-data. Researchers generally impose this as a requirement that software processes support reflective behavior [17]. Yet several distinct realizations of state reification have been described in the literature. Process state is reified in APPL/A process programs by relations that are defined and maintained by the process itself [34]. Providing this support imposes an extra burden on process programmers, but it allows state reification to be tailored precisely to particular process needs. The ProcessWall [18] defines a process state service that can be used by processes where the process language does not provide built-in support for state reification. In SLANG [3, 5], the execution state of Petri nets is maintained in language-defined data structures, which may be accessed by executing processes, for example, for purposes of process modification. All of these approaches satisfy the requirement on software processes to support state reification, and all afford a variety of benefits to justify their use in different circumstances, but only one actually imposes the requirement that a process language support reification.

The fact that process characteristics do not completely define or restrict process language characteristics affords a degree of freedom that enables processes and process languages to be studied and applied more or less independently. This allows a range of process languages to be explored, and it admits experimentation with alternative strategies for the use of those languages. This exploration and experimentation are important because each approach has different costs and benefits. (This argues against the use of fourth-generation languages, which may restrict implementation choices and flexibility.) Ultimately, process language requirements should be determined and evaluated in the context of an overall strategy for process support. Such a strategy should include process engineering methods that specify how the process language should be used to address process needs. In this regard, the meta-processes in which process languages are used will also help to shape process language requirements, perhaps as much as the production processes that the languages are used to represent. The requirements on process languages must be consistent with their intended use according to these overall strategies for supporting process needs.

Throughout the remainder of this section we will distinguish between process and language requirements in our analysis.

2.2 Process Representation and Execution Requirements

Satisfying the goals of representing and executing software processes requires a core set of process-language capabilities. Although several researchers have proposed capabilities for this purpose, there has been little agreement on what capabilities the core should include. For example, Conradi and Liu [11] suggest that this set should contain activities, artifacts, roles, users and groups, production tools, and support for evolution. Junkermann *et al.* [22] explicitly define the set to include activities, roles, artifacts, and resources, but also implicitly include support for artifact interrelationships. Lonchamp [25] indicates that activities, artifacts, agents, roles, tools, and constraints are the "classical" process concepts. While there is some agreement on the inclusion of activities and resources, clearly, there is less consensus on other capabilities.

Analyzing several languages and software processes leads us to believe that the core set should include:

- Activity descriptions: to define the steps that occur as part of a process
- Artifact descriptions: to define the software system under construction, including the implementation code and all of the other artifacts produced
- Resource descriptions: to specify human and computer resources available for use in activities

- Relationships among entities: to indicate various kinds of semantic interconnections among activities, artifacts, and resources; for example, to indicate that one activity may precede another, one artifact (e.g., a test case) may be derived from another (e.g., a requirements specification), one activity may modify an artifact and use a resource

- Consistency management: to ensure the satisfaction of required conditions over interconnected activities, artifacts, and resources and across activities; for example, consistency may be defined in terms of well-formedness constraints over artifacts and order dependencies among activities.

This set seems to satisfy most of the requirements we have seen other researchers impose; for example, it supports roles as special kinds of relations among activities, and users and tools as special kinds of resources (as suggested in [22]). In fact, all of these features can be found in some process languages, and many languages attempt to provide at least minimal support for most of these capabilities. For example, Adele [6], AP5 [8], APPL/A [34], EPOS [10], and Merlin [22] provide at least some support for most or all of the capabilities. This supports the assertion that all of these aspects must be addressed to obtain a complete production process description.

No existing system supports all of the required capabilities sufficiently, however. Typically, process languages have focused on either activity definition or artifact description, and on the corresponding subsets of consistency and interrelationship definitions. This results in corresponding weaknesses in production processes. Although space constraints prohibit extensive elaboration, we discuss some of these limitations below.

Activities and Artifacts: Production processes, like most software systems, must include descriptions of the activities they comprise and the artifacts (data) they manipulate. Unlike many other kinds of software, however, activities in production processes have two complementary but distinct aspects: *proactive* and *reactive* control. Proactive control is what drives the process forward toward a goal. Reactive control is how a process responds to contingencies. Rather than being exceptional occurrences, contingencies arise as a matter of course in production processes, so significant parts of processes are dedicated to describing how to handle them.

Both reactive and proactive control are essential for production processes, yet many process languages focus on only one kind of control. For example, rule-based systems, such as Marvel [23] and Merlin, support reactive control well, but they can only simulate proactive control via appropriate programmer-defined pre- and post-conditions, which is difficult and awkward. State-based and net-based languages, such as Teamware [37], Process Weaver [13], and SLANG [4], support proactive control, but they do not readily support reactive control. As a result, they are awkward to use for describing reactions to contingencies.

Resources: Production processes make use of many different kinds of resources. Resources include, for example, humans involved in the execution of the process and tools used to aid in carrying out activities. Different resources have different attributes; for example, a human resource might have access privileges and number of hours available for a project, while a tool might be characterized in terms of preconditions to be satisfied prior to its use, effects it will have on its operands, etc.

Although resources are a very important part of production process description and execution, few systems support their definition explicitly. For example, AP5, APPL/A, Hakoniwa [21], HFSP [35], Marvel, Pleiades [36], and SLANG provide no predefined mechanisms for resource definition (beyond their artifact and activity definition formalisms). The few systems that do provide some degree of support typically provide descriptions only of human resources and tools (e.g., Teamware and EPOS). The ability of such systems to accommodate more complex resource models remains to be demonstrated. For example, how readily will such systems be able to accommodate necessary distinctions between resources that are shared vs. exclusive, consumable vs. non-consumable, active vs. passive, concrete vs. logical, preemptable vs. non-preemptable, etc.? No resource model of which we are aware addresses all these issues in the representation and use of resources by software processes.

Relationships and Consistency: Production processes must be able to capture and manipulate many different kinds of relationships among activities, artifacts, and resources. Among some of the more commonly used kinds of relationships are *inter-activity* (e.g., composition and execution order dependencies), *inter-artifact* (e.g., dependency relationships), and relationships between activities and the artifacts they manipulate, between activities and the resources they use, and between resources and the artifacts they use or modify. These relationships, for example, affect the definition and planning of a process by specifying valid, invalid, and required sequences of activities and reactive responses (which can be inferred based on relationships among activities and between activities and artifacts or resources). Many kinds of relationships also serve as a basis for change impact analysis and consistency management.

In conclusion, we note that a variety of paradigms that have been found useful in conventional application programming likewise seem useful in supporting many aspects of process representation and execution.

2.3 Granularity and Detail

Different aspects and kinds of production processes require different amounts of information about process activities, artifacts, and resources. For example, a managerial process might need to know which developers are assigned to which projects, and how many of their hours are already committed. This requires knowing few details beyond the names and work schedules of developers. A process supporting a developer, on the other hand, might need to know precisely which function in a module the developer modified, which other modules depend on that particular function, and how they depend on it. Knowing these details allows the process to assess accurately the impact of the modification on other artifacts and to plan precisely the propagation of those changes. It also ensures that effort is not wasted—for example, developers can be told exactly where to concentrate their efforts, and if a change does not actually affect anything crucial, it need not result in resources being allocated to propagate the effects of the change. Production processes thus require support for the entire spectrum from coarse- to fine-grained representations of activities, artifacts, and resources. They also require connections among different levels of granularity, to facilitate "zoom-in" and "zoom-out" as necessary.

Most existing process languages support both coarse- and finer-grained descriptions of activities, though the treatment varies. For example, ProSLCSE [26] supports activity decomposition to any level of detail, but the decompositions have limited semantics, so few of the benefits of including detailed information (e.g., analysis, planning) can be realized. SLANG [4] supports decomposition down to the level where tools are invoked, but it cannot describe tools and their effects, so it does not provide deep enough semantics to elaborate activities fully and must, consequently, include an "escape" mechanism to invoke tools that are defined elsewhere. Connections among higher- and lower-levels of activity descriptions have also received varying treatment. Some languages, like MVP-L [33], HFSP [35], SLANG, and APPL/A [34], support explicit activity decomposition, and thus, it is relatively straightforward to move between levels of granularity. Other languages, particularly rule-based formalisms like Marvel [19] and Merlin [22], allow activities to be described in detail and define deep semantics for those definitions, but the activity hierarchy must be extracted from the rule base, which is non-trivial.

Existing process languages have tended to be fairly bi-polar on the issue of granularity of artifact descriptions. Many languages support the definition of "proxies" to represent artifacts. Proxies are essentially black boxes that represent a discrete artifact. They are distinct from the artifact itself. Proxies have associated attributes that summarize any details of their corresponding artifacts that are important to the processes that manipulate the artifacts. The processes cannot themselves access the actual artifact, except through invoked tools, and tools cannot access the proxies. This approach has been adopted in Marvel, Merlin, SLANG, and GRAPPLE [20]. Other languages support the definition of artifacts at any level of granularity. For example, APPL/A, Adele [6], AP5 [8], and Pleiades [36] are based on general-purpose programming languages, and they provide rich type models that support the fine-grained definition and manipulation of artifacts. Treatment of connections among higher- and lower-levels of activity descriptions has been variable. Several languages support "is-composed-of" and other types of structural or dependency relationships among artifacts (e.g., Adele, EPOS [10], Marvel, and Merlin). The type models in APPL/A, AP5, and Pleiades support arbitrarily complex composition and decomposition of data types.

In general, while it is possible to abstract away details from a fine-grained representation if a particular process does not need them, it is not possible to synthesize additional details from a coarse-grained representation when a process does need them. Further, details are often necessary for deciding how the process should proceed, for making predictions about future process behavior, for assessing past process performance, etc. Languages that do not support the detailed expression of process elements (activities, artifacts, resources) provide correspondingly limited support for production processes.

When a language supports only coarse-grained representations, it is limited, a priori, to providing coarse-grained interrelationships and consistency definitions (e.g., Marvel and Merlin). Many kinds of fine-grained relationships convey deep semantics. For example, a "depends-on" relationship from a design document to a requirements document can indicate that the documents are mutually inconsistent, but relationships between specific design elements and the requirements they satisfy can indicate precisely which parts are inconsistent and how. Similar arguments can be made for interrelationships among activities, between activities and artifacts, etc. Thus, it is important for processes to be able to capture relationships among activities, artifacts, and resources at different levels of granularity. We note that these same lessons have also been learned about languages for software product development.

2.4 Visualization

Humans can often absorb complex information efficiently if it is represented pictorially. As software processes are notoriously complex, and because a grasp of this complexity can be of great importance, there has been considerable attention paid to support for visualization in process languages. The specific goals of process visualization include

- Improved intuition about process: Superior pictorial representations can enhance human intuition. For example, it has been hypothesized that process visualizations can help humans to more accurately infer where processes can be parallelized, where iterations occur, and where execution bottlenecks might be expected. By helping humans infer these important process characteristics, such visualizations also support more accurate reasoning about processes.

- Surer sense of process execution status: Superior visualizations can enhance understanding of the static and dynamic properties of processes. Thus good visualizations can help project personnel understand such key situations as the state of project completion, the extent to which resources and personnel are being used effectively, and whether progress is acceptable.

- Improved communication among team members: It has been hypothesized that superior process visualizations help project members communicate about key project issues. Such visualizations can be used to indicate when coordination efforts are needed, and on what subjects, and to help avert misunderstandings.

- Ease of modification: Software processes routinely execute for years, so they are likely to require modification during their execution. The complexity and subtlety of software processes severely aggravate the difficulty of such changes. Changes must be carefully considered and well supported by good insights and intuitions. Thus, superior pictorial visualizations of such processes may be important assets for process modification.

One common approach to process visualization, which was taken in IDEF0 [32], Process Weaver [13], and Teamware [37], is to use a single pictorial representation to define software processes. Most of these representations depict process activities and their interrelationships as graphs with nodes (activities) and edges (interrelationships). Some representations provide different depictions to show different kinds of process activities and interrelationships.

Virtually all visualization systems seem to acknowledge that users should not be confronted with very large representations. Too much or too many kinds of information on one diagram seem to hinder at least some types of comprehension. Thus, for example, IDEF0 is often used to show only dataflow between process activities. When it does this, and nothing further, a very clear picture often emerges. IDEF0 can also be used, however, to depict other interrelationships simultaneously, which results in a far more cluttered and confusing picture and the loss of many benefits of visualization (clarity, enhancement of intuition, etc.). A similar phenomenon is noted in the use of Petri nets, which have become a fixture for clearly depicting flow and synchronization of activities. In systems such as SLANG [4] we see that Petri nets function very successfully in depicting these aspects of software processes. In SLANG, however, we also see that, as the basic Petri net vehicle is elaborated (e.g., by adding colored tokens and specialized transitions), the result is increasing complexity but decreasing comprehensibility.

One attack on the self-defeating clutter of overly elaborate diagrams has been to present process visualizations as coordinated sets of diagrams. Thus, systems such as Statemate [15] and Oikos [28] represent processes using multiple diagrams, each presenting a different view of the process. With such systems users can browse a variety of different diagrams, each of which represents a focused view.

A major problem with such approaches is the need to maintain consistency among multiple views. Systems such as Oikos address this problem by generating all views from a single common non-pictorial representation of the process. Thus, the view-generators keep all of the views consistent with the common representation and with each other. Another key advantage of this approach is that it eases problems of providing a strong semantic basis for the visualization. With primitive visualization systems such as IDEF0, the depicted relations are defined rather informally. This can lead to misinterpretations of the diagrams. Systems such as SLANG, which base the Petri net-like depictions upon rigorously defined semantics, do not suffer from this problem. The Oikos approach seems to combine the advantages of simple, clear diagrams, multiple views, and strong semantic basis for the views.

From this brief survey of visualization efforts we can infer the following:

- Pictures improve communication and help intuition: Process visualizations aid in communication by distilling considerable amounts of information. These visualizations can be highly suggestive, supporting useful, important inferences.
- Pictures work best when they depict modest amounts of information: As the quantity and diversity of information covered by the depiction increases, intuition and comprehension decreases.
- Pictures can be ambiguous and may promise more than they deliver: In the absence of a semantic basis for a visualization, different users can draw different conclusions from the visualization, leading to conflict and confusion. Moreover, while visualizations stimulate users' intuition, there is a danger that the inferences drawn by users may be incorrect. The issue here is the solidity of the semantic base from which the visualizations are drawn, and the extent to which users reliably draw correct inferences from the visualization.
- Multiviewers help with visualization problems: The use of a semantically deep formalism as the basis for the drawing of a coordinated set of views seems to support most of the desirable properties of visualization while avoiding many of the problems.

- Strong, non-visual representation is of central importance: The foregoing has lead us to conclude that, while visualizations can be most beneficial in facilitating important intuitions, they are most useful and reliable when they are drawn from a semantically rich non-visual representation.

These conclusions suggest that visualizations function best when they complement semantically rich, non-visual representations. We believe that this synergy is directly analogous to the synergy between software designs or models and code. Further we suggest that this analogy should guide the process community towards a more useful understanding of how process models and process code should be used together.

2.5 Language Architectures for Process Support

In Section 2.2, we reviewed the aspects of software processes that we believe are essential for process execution. These aspects must be addressed through process languages. In this section we discuss alternative approaches that provide support using one or more languages.

One Semantically-Broad Language: One approach to supporting software processes is through a single, semantically broad language. Examples of such languages include APPL/A [34], Merlin [22], Adele [7], and AP5 [8]. This approach has several benefits. The use of a single language provides comprehensive and integrated syntax and semantics across multiple process aspects and subdomains. In turn, one language is usually easier to learn, understand, analyze, and use than multiple languages.

On the other hand, a single large language may be very large indeed, and even so it is unlikely to support all of the features and approaches that may be useful in different software processes. The generality of such a language may be achieved by lowering the semantic level of the language towards general-purpose features, as in APPL/A, but this can complicate understanding and analyses at the high level (see Section 2.6). Alternatively, generality may be achieved by taking the "union" of a variety of high-level abstractions, which can lead to further complexity and interoperability problems without guaranteeing generality.

Multiple, Independent, Special-Purpose Languages: An alternative to one broad language is the use of multiple, semantically specialized languages [30]. The advantage of this approach is that specialization within an individual language allows simpler, more concise, more analyzable, and more understandable programs within that particular domain.

Additionally, the availability of a variety of such languages can allow a choice of the most appropriate language or model for a particular purpose within a process. When multiple languages are used, however, the issue of their uniformity, understandability, and analyzability as a group is an issue. Additionally, the interoperability of the different languages and their respective models for control, data, consistency, etc., becomes a problem.

Core Languages with Extensions: Conradi and Liu [11] identify a third alternative, that is, the use of a common "core" language extended with higher-level and more specialized languages. This approach should be flexible enough in principle to support special process needs or domains in a relatively straightforward manner. However, defining the core language is a problem on the order of defining a single, broad process language. It also raises ongoing questions about whether each capability belongs in the core or in an extension. As observed in [11], for example, putting support for meta-processes into the core can be inefficient (since the execution of a core-level meta-process requires translation to affect the various extensions); however, putting it into the extensions means that capabilities may be repeated among the extensions and more difficult to coordinate and maintain. In general the approach will require translation between the core language and the various extensions; this problem may or may not be more difficult that the corresponding interoperability problem when multiple languages are used independently (as described above).

General Implications for Process Languages: Languages that attempt to "do everything" for a process are probably unworkable because of their size and complexity. Software process is a domain in which many difficult problems occur. However, attempting to provide direct support to solve all of those problems leads to "kitchen sink" languages that may be powerful but are too difficult to use. It seems to us that a more viable alternative is to define somewhat simpler languages that can easily accommodate the more normal cases but are general enough to admit the programming of alternative approaches to special cases.

Each of the three approaches described above seems plausible in principle, and experience has already shown that each also suffers from limitations of one sort or another. Approaches based on a core language do not seem to be essential, but neither should they be excluded. Montangero [27] argues that the multiple-language approach should be pursued over the shorter term to identify requirements on suitable core languages for the longer term. There seems to be no single correct answer to the question of how linguistic support for software processes should be structured. More work is warranted on all approaches.

We also note that designers of classical programming languages have struggled with similar issues and approaches over the decades—with a similar lack of resolution. However, these decades of experience do not seem to us to be adequately exploited by process language designers.

2.6 A Semantic Space of Process Languages

Different kinds of activities related to software processes may be supported more or less well by different kinds of process languages. When defining or choosing a process language, it is important to have a means for determining what sort of language best suits the intended process application. Towards this goal, we have found it useful to characterize process languages in terms of three general semantic properties: *semantic breadth* refers to the range of support for various aspects of software processes, *semantic level* refers to how closely a language's abstractions match the concepts in software processes, and *semantic depth*, by which we mean semantic formality, rigor, precision, and completeness.

These properties represent the dimensions of a semantic space that includes all process languages. Points throughout the space add value for some aspects of process representation and execution and help to distinguish the uses for which particular languages are appropriate. A variety of positions in the semantic space are taken by current process languages. Some representative examples are:

- ProSLCSE [26], IDEF-0 [32], Process Weaver [13], and Teamware [37] help to orchestrate, and may partially automate, process tasks and tools. They have a high semantic level, but their breadth is narrow, even in describing process activities. Their depth ranges from shallow to moderate, since they admit few formal analyses and, if they support process execution, it is to a limited extent.
- MVP-L [33] is intended specifically for process modeling. It necessarily has a high semantic level, medium semantic breadth, and shallow semantic depth, and it is not intended to support execution.
- APPL/A [34], Merlin [22], Marvel [23], and EPOS [10] are intended to produce fully executable process programs. They require the ability to support a broad range of functionality, with semantics sufficiently deep to enable execution. Their level of abstraction is generally low to moderately high, though Adele-TEMPO, which also supports executable process programs, combines a low-level and high-level language [6].

An analysis of language suitability for various users and roles in a software process [33] supports the contention that the appropriate applications of process languages are determined by characteristics related to semantic level and breadth.

This semantic space provides a framework for analyzing the suitability of languages for particular sorts of roles in support of process representation and execution. The same semantic dimensions characterize specification languages, design formalisms, and coding languages that are used in conventional software development. The place of various languages in the software product life cycle is usually well recognized. A similarly clear view of the roles of various languages in the software process life cycle remains to be achieved.

2.7 Process Meta-Capabilities

Software processes require a variety of *meta-capabilities.* These are capabilities that reify, affect, or control the use of other capabilities.[2] For processes, we believe that meta-capabilities should apply to the core process functionalities, namely, activities, artifacts, resources, relationships, and consistency. We see three main categories of meta-capabilities that software production processes require:

- *Meta-data* describe the status or state of entities in a software process, e.g., the execution state of tasks, the types of objects, the usage of resources, the enforcement status and state of constraints, etc.

- *Reflectivity* is the ability of a software process to assess its own properties (e.g., execution state) and act on those assessments. Reflectivity is useful in software processes for a wide variety of purposes, such as planning activities based on projected resource availability, making control decisions based on the consistency of artifacts, maintaining artifact consistency based on evolving versions of type definitions, and so on.

- *Dynamism* is the ability to make decisions about and changes to, a process during its execution. Typically dynamism is considered to affect fundamental aspects of the process definition or attributes. The decisions may range, for example, from selection among alternative activities, to the modification of existing activities, to the addition and deletion of activities. Other sorts of dynamism include, for example, the ability to make dynamic adjustments of resource allocations and the ability to dynamically control required consistency conditions. Dynamism may be exercised from within a process, constituting a form of reflectivity (for example, if a process is self-modifying). Dynamism may also be imposed from external sources, e.g., when a process-maintenance process operates to make improvements in an ongoing production process.

[2]Evolution is typically defined as a meta-capability as well, but we believe it to be separable. See Section 2.8 for a discussion.

Meta-data, reflectivity, and dynamism have been supported in various ways and for different purposes. APPL/A [34] programs typically include some programmed representation of activity and artifact state, defined using more general-purpose constructs. These are typically used for dynamic process control. Pleiades [36] provides dynamic information about artifact type and instance definitions, and dynamic control over several object management capabilities, including persistence and the enforcement of constraints on artifacts. These capabilities are used, for example, to reason about the states of artifacts and to plan modifications to them accordingly. AP5 [8] takes a meta-data approach to the dynamic accommodation of artifact inconsistencies [1]. AP5 and Merlin [22] define rules as data, enabling them to be manipulated reflectively and dynamically. SLANG [3] and EPOS [10] also define manipulable data representations for process models. These are especially intended to support process evolution.

Support for meta-data, reflectivity, and dynamism are requirements on most software processes. Many different kinds of meta-data may be useful in software processes. However, there is probably a limit to the amount of meta-data that a language or process can effectively maintain and apply. More research is needed to determine (among other things) the kinds of meta-data that are most useful to production processes, the most appropriate strategies for the reflective use of meta-data, and the extent to which process-independent capabilities provided at the language level can support process-dependent semantics at the process application level. A major language design issue is the kinds and extent of dynamism that should be supported or accommodated. Too little linguistic dynamism complicates the programming of dynamic processes, but too much linguistic dynamism undermines process stability (and thus understandability, analyzability, predictability, reusability, and so on). Finally, it should be noted that dynamic behavior in processes is not wholly dependent on dynamic capabilities in process languages; it may be supported in a variety of ways. This is a particular application of the general principles enunciated in Section 2.1 and elaborated in Section 2.8.

2.8 Meta-Processes

An important distinction in software processes is between *production processes* and *meta-processes* [3, 9, 35]. Production processes, which are the focus of this paper, support the development and maintenance of software products. Meta-processes are essential to production processes, since meta-processes effect the creation, execution, management, and evolution of production processes. A goal for production processes is to be readily, yet safely, manipulable by meta-processes, e.g., for process maintenance and evolution. A goal for process languages is to facilitate the systematic and principled application of meta-processes to production processes according to sound software-engineering practices.

The predominant approach taken so far to specifying meta-processes has been to define process languages in which support for production processes and support for meta-processes are combined. A typical technique for supporting process evolution in these languages is to represent the production process fundamentally in the form of data, and then effect process evolution by modifying the data representations. Examples of this approach are EPOS [10], in which production processes are represented as a combination of task networks and rules, SLANG [4], which represents production processes as modified Petri nets, GRAPPLE [20], which represents production processes as plans, and Merlin [22], which represents processes as mutable rules.

For our purposes, the approach of combining representation of production processes and meta-processes complicates the representation of production processes. It yields process representations that combine code and data for meta-process activities with code and data that represent production-process activities. the software product. The complication of production-process representations is especially problematic because production processes are themselves complex, and process programs to support them are correspondingly complex. This places a premium on their understandability, verifiability, analysis, reusability, etc. These qualities are all harder to achieve in a process program if the code and data of the meta-process must be "filtered out" in order to make the representation of the production process visible. Our belief is that the considerable weight of software development experience should guide the process community here. Given the complexity of both production processes and meta-processes, a separation of these concerns is desirable for the sake of simplification.[3]

The evolution technique that has been most used, i.e., representing production processes as data, has an additional drawback. That is, those data must be interpreted to understand the production process, but the data are subject to change. The process-program code is primarily for the meta-process, so analysis of that code conveys little information about the production process. While this technique may be suitable in some cases, it is only one of several that may be used to support process evolution. Other techniques include, for example, type and data evolution (e.g., [24]), late and dynamic binding, dynamic linking, tool buses [31], object request broker architectures [12], and programming techniques, such as subjectivity [16], object roles in Adele [6], and program design techniques to support evolution, like intermittent execution [34]. All of these techniques have benefits and may be

[3] "Separation of concerns is a common-sense practice that we try to follow in our everyday life to master the difficulties we encounter. The principle should be applied also in the case of software development, to master its inherent complexity" ([14], p. 47).

more appropriate for various purposes than using data representations for processes.

We believe that research into both processes and meta-processes should continue. However, because these processes are subject to different and even competing requirements, we believe this research should proceed largely on parallel, not overlapping, tracks. Languages for production processes in particular should benefit from a focus on issues that affect software production, since there is still much to be learned there.

3 Conclusions

The foregoing sections include a number of observations about particular aspects of process language definition and and use. These allow inferences about the suitability of particular kinds of languages and language characteristics for particular purposes in the software process life cycle. The use of process languages in support of software processes has been successful in a variety of ways. This success leads us to believe that the approach remains viable and promising. Furthermore, it allows us to begin to validate the hypothesis that there are strong parallels between techniques and issues of software development, and key techniques and issues in process development. In this section, due to space limitations, we can only address some of the parallels, and select those that seem the most compelling.

Importance of Code and Non-Code Representations: In software development there has long been a recognition that a final product must be much more than naked code. Design, architectures, requirements, and evaluative material are essential as well. This variety of artifacts is necessary to assure that needs other than the need to execute on a computer can be met. The same seems to be true in the process domain. Process code (such as APPL/A [34], Marvel [23], etc.) supports process execution, but there are other needs as well (e.g., group coordination) that are better met by design artifacts such as process models. Thus, the steady growth of interest in process modelling, process architecture, and process measurement tools and technologies seems expectable. What is unexpected is the continual disappointing lack of interest in process requirements technology. This obviously key technology area should be attracting aggressive attention.

Diversity of Representation Language Paradigms: Currently there is considerable experimentation with a wide variety of languages and language paradigms in an attempt to determine which are most successful in effectively representing process information. No clearcut winners or losers have been identified. This also closely parallels decades of experience with design and programming language research. These decades of research have still failed to produce a single general-purpose language that has been generally accepted by the design and programming communities. We suggest that process development does indeed from this perspective appear to be tantamount to programming in a specialized application area (software process), but that area is sufficiently broad that different languages are likely to be more or less effective in representing different types of processes.

There is growing evidence, however, that the situation in the process domain may be even more difficult than in other, more traditional application areas. Research to date has clearly indicated that process languages need to be reflective and dynamic (to name just two troublesome characteristics) for reasons that are now increasingly clear. Thus process languages seem to require more semantic features than seem to be sufficient for most other applications areas. Thus, research in this area is likely to stretch the current limits of programming language research in particularly challenging ways. More research is needed in order to better understand and specify what is needed. This should help determine where existing language research can help, and where new ground must be broken.

Importance of Semantics: In software development the importance of a strong semantic basis for architecture and design specifications is increasingly understood to be essential, as this permits safe and effective reasoning and inference about these key precode artifacts. In a similar way, experience to date indicates that process models benefit from a strong semantic basis, and suffer from the lack of one. Both software design diagrams and process models strongly heighten intuition when they are clear and suggestive. In the case of software designs, however, the lack of semantics prevents effective checking for such errors as race conditions, dataflow anomalies, and possible deadlocks. The process community seems to be just starting to realize that process models that are only pictures, not based upon precise semantics, are open to varying interpretations, none of which can or should be trusted.

Evolution is Crucial, but Difficult: In both software development and process development it now seems clearly understood that the end product cannot be regarded as fixed, final, and unvarying. As both sorts of products interact with a changing world, they will likewise need to undergo continuous change. The difficulty of safely and effectively evolving software products is currently a source of continuing frustration, and the target of continuing research. In the domain of process development, the analogous sort of work is referred to as meta-process research, and is no less important and troublesome. Here, however, there seems to be a clear difference in approaches. Meta-process work seems to be far more inclined to countenance evolution of the process while it is actually executing, and often using formalisms that are the same as, or very similar to, process formalisms themselves. In contrast, software is rarely evolved while it is actually executing. Further, while there is a belief that there are key kinships between software evolution technologies and software development technologies, there seems to be little confidence that current technologies in one area will adequately address the problems in the other area. These differences may prove to be fundamental. It is likely that the process domain will demand that technologies for evolving running processes safely will need to be developed. On the other hand, it is far less clear that a uniform formalism and technology for both process development and process evolution can be found or should be sought. Further research on these subjects is clearly needed.

Closing Thoughts: Continually accumulating experience indicates that process development is profitably thought of as the development of a sort of software—that it is essentially programming in a specialized domain. Software development analogies have guided numerous process technology development efforts, often to beneficial outcomes. The software development idiom seems to be a likely source of inspiration for important new process research (e.g.,. process requirements formalism is a promising research area), and should be reassessed from this point of view. In some cases, however, the analogy seems to be less valid. For example, reflectivity, and technologies for evolving running systems, seem to characterize the process domain and distinguish it from software development. Development of these technologies for process will probably find little programming technology to exploit, but are likely to return technology that will prove useful to software development. Thus we continue to believe that the search for similarities and differences between software and process development has been, and will continue to be, a lively and mutually beneficial enterprise.

Acknowledgements

This work has benefited from the ideas and contributions of several people. Rodion Podorozhny helped to identify some of the important characteristics of such resources. The students in the Spring, 1995 seminar on software processes provided many insights on the problem of supporting software processes. Finally, the authors would like to thank their colleagues in the Arcadia Consortium for many years of feedback and ideas. Dennis Heimbigner has made particularly valuable contributions to our ongoing process research.

4 References

[1] Robert Balzer. Tolerating inconsistency. In *Proc. of the 13th International Conference on Software Engineering*, pages 158 – 165, May 1991.

[2] Sergio Bandinelli, Elisabetta Di Nitto, and Alfonso Fuggetta. Policies and mechanisms to support process evolution in PSEEs. In *Proc. of the Third International Conference on the Software Process*, pages 9–20, 1994.

[3] Sergio Bandinelli and Alfonso Fuggetta. Computational reflection in software process modeling: the SLANG approach. In *Proc. of the 15th International Conference on Software Engineering*, pages 144–154, 1993.

[4] Sergio Bandinelli, Alfonso Fuggetta, Carlo Ghezzi, and Luigi Lavazza. SPADE: An Environment for Software Process Analysis, Design, and Enactment. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, chapter 9, pages 223–248. Research Studies Press, Ltd., Taunton, Somerset, England, 1994.

[5] Sergio Bandinelli, Alfonso Fuggetta, and Sandro Grigolli. Process modeling in-the-large with SLANG. In *Proc. of the Second International Conference on the Software Process*, pages 75–83, 1993.

[6] Noureddine Belkhatir, Jacky Estublier, and Walcelio Melo. ADELE-TEMPO: An Environment to Support Process Modelling and Enaction. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, chapter 8, pages 187–222. Research Studies Press, Ltd., Taunton, Somerset, England, 1994.

[7] Noureddine Belkhatir, Jacky Estublier, and Melo L. Walcelio. Software Process Model and Workspace Control in the Adele System. In *Proc. of the Second International Conference on the Software Process*, pages 2 – 11, 1993.

[8] Don Cohen. *AP5 Manual.* Univ. of Southern California, Information Sciences Institute, March 1988.

[9] Reidar Conradi, Christer Fernstrom, and Alfonso Fuggetta. Concepts for Evolving Software Processes. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, chapter 2, pages 9–31. Research Studies Press, Ltd., Taunton, Somerset, England, 1994.

[10] Reidar Conradi, Marianne Hagaseth, Jens-Otto Larsen, Minh Ngoc Nguyen, Bjorn P. Munch, Per H. Westby, Weicheng Zhu, M. Letizia Jaccheri, and Chunnian Liu. EPOS: Object-Oriented Cooperative Process Modelling. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, chapter 3, pages 33–70. Research Studies Press, Ltd., Taunton, Somerset, England, 1994.

[11] Reidar Conradi and Chunnian Liu. Process Modelling Languages: One or Many? In Wilhelm Schäfer, editor, *Proceedings of the 4th European Workshop on Software Process Technology*, pages 98–118. Springer-Verlag, 1995. Published as Lecture Notes in Computer Science, number 913.

[12] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design, Inc., and SunSoft, Inc. *The Common Object Request Broker: Architecture and Specification.* Object Management Group and X/Open, 1993. Revision 1.2.

[13] Christer Fernström. PROCESS WEAVER: Adding process support to UNIX. In *Proc. of the Second International Conference on the Software Process*, pages 12 – 26, 1993.

[14] Carlo Ghezzi, Medhi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1991.

[15] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. on Software Engineering*, 16(4):403 – 414, April 1990.

[16] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 411–428, 1993.

[17] Dennis Heimbigner. Experiences with an Object-Manager for A Process-Centered Environment. In *Proceedings of the Eighteenth International Conf. on Very Large Data Bases*, Vancouver, B.C., 24-27 August 1992.

[18] Dennis Heimbigner. The ProcessWall: A Process State Server Approach to Process Programming. In *Proc. Fifth ACM SIGSOFT/SIGPLAN Symposium on Software Development Environments*, Washington, D.C., 9-11 December 1992.

[19] George T. Heineman, Gail E. Kaiser, Naser S. Barghouti, and Israel Z. Ben-Shaul. Rule Chaining in MARVEL: Dynamic Binding of Parameters. *IEEE Expert*, 7(6):26–32, December 1992.

[20] Karen E. Huff and Victor Lesser. A plan-based intelligent assistant that supports the software development process. In *ACM Symposium on Practical Software Development Environments*, pages 97 – 106, 1988.

[21] H. Iida, K.-I. Mimura, K. Inoue, and K. Torii. Hakoniwa: Monitor and Navigation System for Cooperative Development Based on Activity Sequence Model. In *Proc. of the Second International Conference on the Software Process*, pages 64 – 74, 1993.

[22] G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf. Merlin: Supporting cooperation in software development through a knowledge-based environment. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, chapter 4, pages 103–129. Research Studies Press, Ltd., Taunton, Somerset, England, 1994.

[23] Gail E. Kaiser, Naser S. Barghouti, and Michael H. Sokolsky. Experience with Process Modeling in the Marvel Software Development Environment Kernel. In Bruce Shriver, editor, *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 131–140, Kona HI, January 1990.

[24] Barbara Staudt Lerner. Type Evolution Support for Complex Type Changes. Technical Report TR–94–71, University of Massachusetts, Computer Science Department, Amherst, MA, October 1994.

[25] Jacques Lonchamp. An assessment exercise. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, chapter 13, pages 335–356. Research Studies Press, Ltd., Taunton, Somerset, England, 1994.

[26] James Milligan. The process-oriented software life cycle support environment (ProSLCSE). briefing slides, 1994. Rome Laboratory/C3CB, Griffiss Air Force Base, N. Y. 13441-4505.

[27] C. Montangero. In favor of a coherent process coding language. In Wilhelm Schäfer, editor, *Proceedings of the 4th European Workshop on Software Process Technology*, pages 94–97. Springer-Verlag, 1995. Published as Lecture Notes in Computer Science, number 913.

[28] Carlo Montangero and Vincenzo Ambriola. OIKOS: Constructing process-centered sdes. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, pages 33 – 70. John Wiley & Sons Inc., 1994.

[29] Leon J. Osterweil. Software Processes are Software Too. In *Proceedings of the Ninth International Conference of Software Engineering*, pages 2–13, Monterey CA, March 1987.

[30] Leon J. Osterweil and Dennis Heimbigner. An Alternative to Software Process Languages. In C. Ghezzi, editor, *Proceedings of the Ninth International Software Process Workshop*, Airlie, VA, 1994.

[31] James M. Purtilo. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, 1992.

[32] Richard J. Mayer, et al. IDEF Family of Methods for Concurrent Engineering and Business Re-engineering Applications. Technical report, Knowledge Based Systems, Inc., 1992.

[33] H. D. Rombach and M. Verlage. How to assess a software process modeling formalism from a project member's point of view. In *Proc. of the Second International Conference on the Software Process*, pages 147 – 159, 1993.

[34] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. APPL/A: A language for software-process programming. *ACM Trans. on Software Engineering and Methodology*, 4(3), July 1995. to appear.

[35] Masato Suzuki and Takuya Katayama. Meta-operations in the process model HFSP for the dynamics and flexibility of software processes. In *Proc. of the First International Conference on the Software Process*, pages 202 – 217, 1991. Redondo Beach, California, October, 1991.

[36] Peri L. Tarr and Lori A. Clarke. PLEIADES: An Object Management System for Software Engineering Environments. In *ACM SIGSOFT '93 Symposium on Foundations of Software Engineering*, pages 56–70, Los Angeles, December 1993.

[37] Patrick S. Young. *Customizable Process Specification and Enactment for Technical and Non-Technical Users.* PhD thesis, University of California at Irvine, 1994.