APPL/A: A LANGUAGE FOR SOFTWARE-PROCESS PROGRAMMING

Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil

CU-CS-727-94

APPL/A: A Language for Software-Process Programming

CU-CS-727-94 1994

Submitted to ACM Transactions on Software Engineering and Methodology, June, 1994

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

APPL/A: A Language for Software-Process Programming [*]

Stanley M. Sutton, Jr.   Dennis Heimbigner   Leon J. Osterweil[†]

Department of Computer Science
University of Colorado
Boulder, CO 80309-0430

Abstract

Software-process programming is the coding of software processes in executable programming languages. Process programming offers many potential benefits, but their realization has been hampered by a lack of experience in the design and use of process-programming languages. APPL/A is a prototype software-process programming language developed to help gain this experience.

APPL/A is intended for the coding of programs to represent and support software-processes including process, product, and project management. APPL/A is defined as an extension to Ada, to which it adds persistent programmable relations, concurrent triggers on relation operations (for reactive control), optionally and dynamically enforcible predicates on relations (which may serve as constraints), and composite statements that provide alternative combinations of serializability, atomicity, and consistency enforcement (for programming high-level transactions).

APPL/A has been used to code engineering-oriented applications, like requirements specification and design, as well as management-related activities, such as personnel assignment, task scheduling, and project monitoring. APPL/A has also enabled us to experiment with process-program design techniques and architectures, including process-state reification, intermittent (or persistent) processes, reflexive and meta-processes, and multiple-process systems. Our ability to address a wide range of software processes and process characteristics indicates that the APPL/A constructs represent important and general capabilities for software-process programming.

Categories and Subject Descriptors: D.1.m [**Programming Techniques**]: Miscellaneous; D.2.2 [**Software Engineering**]: Tools and Techniques; D.3.2 [**Programming Languages**]: Language Classifications; D.3.3 [**Programming Languages**]: Language Constructs and Features; H.2.3 [**Database Management**]: Languages–*database (persistent) programming languages*; I.m [**Computing Methodologies–Miscellaneous**]; J.6 [**Computer-aided Engineering**]; K.6.3 [**Management of Computing and Information Systems**]: Software Management–*software development, software maintenance*

General Terms: Design, Experiment, Languages, Management, Reliability

Additional Key Words and Phrases: Software process programming, software-process programming languages, transaction management, consistency management, multi-paradigm programming languages

[*]This research was supported by the Defense Advanced Research Projects Agency, through DARPA Order #6100, Program Code 7E20, which was funded through grant #CCR-8705162 from the National Science Foundation. Support was also provided by the Naval Ocean Systems Center and the Office of Naval Technology.

[†]Department of Information and Computer Science, University of California, Irvine, CA 92717. Current address: Software Development Laboratory, Department of Computer Science, University of Massachusetts, Amherst, MA 01003.

1 Introduction

The idea of "process" has come to be seen as fundamental to software development. This is indicated by an increasing number of international conferences and workshops dealing with the software process (such as the International Software Process Workshops and the International Conference on the Software Process, among others). A software development process, or software process, can be defined as a set of steps for creating and evolving software systems that encompasses both technical and managerial concerns [18]. Software processes are seen as crucial to software quality and cost [43], and they have been characterized as providing the most practical framework for making improvements in the performance of organizations that develop software [31]. The specification and application of high-quality software processes may also lead to enhanced contributions by computers [49].

The ability to represent and execute software processes is essential to their study and use. One important approach to the representation of software processes is process *modeling* [43, 40, 16]. Software process modeling is the representation of selected aspects of a software process. Some common examples of software-process models are data-flow and control-flow diagrams, statecharts [25], and Petri nets [51]. Process models have been used to support process planning, replanning, monitoring, and recording [41, 46], to represent, analyze, and compare development methodologies [60, 69], to simulate process executions [58, 17], and to provide automated process support and automated process execution (for example, [21, 9, 58, 17, 1]).

Software-process *programming* [48] is a related but contrasting approach. Software-process programming brings a software-engineering paradigm to the development and use of software processes. It represents software processes as programs written in programming languages. A particularly important aspect of the approach is an emphasis on the *executability* of process programs. That is, while a process program may provide a model of a process, it should more importantly automate or otherwise automatically support the execution of the process. In this regard, process programs can be viewed as process *implementations*, which may serve as process models but which have purposes that models may lack [65].

The ability to program software processes depends on the availability of software-process programming languages. When the idea of software-process programming was introduced, however, there were no such languages. It was not clear what these languages would be like, what their requirements might be, or how they would be used. In order to help clarify these issues, we hypothesized requirements for a software-process programming language, defined a prototype language with capabilities and features that addressed these requirements, and experimented with the language by writing software-process programs. This language is called APPL/A.

APPL/A is defined as an extension to Ada [73]. Ada offers the general programming-language constructs that we believe are necessary in a software-process programming language. The main additional constructs that APPL/A provides specifically for software-process programming include relation units with programmable bodies, concurrent trigger units that react to operations on relations, predicate units that may be optionally enforced like constraints on relations, and a set of statements for composite, transaction-like operations on relations. A translator, APT [26], has been developed for a subset of the language. APPL/A has been used to program detailed solutions to the problems posed for the Sixth and Seventh International Software Process Workshops (ISPW6/7) [42, 27], to program an executable prototype requirements-specification process [67], and for a variety of other processes (including programs in the Arcadia [34] prototype demonstration). This paper gives a detailed overview of APPL/A. The paper provides background on language requirements and design, presents the language definition in substantial detail, describes our experience with the language, and discusses the resulting lessons learned about the development and use of APPL/A in particular and software-process programming languages in general.

The organization of the remainder of the paper is as follows. Section 2 discusses issues affecting the requirements for a software-process programming language, gives the corresponding design goals for APPL/A, and describes the approach we took to developing the language. Section 3 presents the language with simple examples, and Section 4 reviews the current status of the language. Section 5 describes various software-process programs that we have written using APPL/A, including particular language constructs and process-programming techniques. Section 6 then presents an analysis and evaluation of language features and related issues. Finally, Section 7 discusses related work, and Section 8 presents a summary and conclusions.

2 Language Requirements, Design Goals and Approach

In attempting to define a prototype software-process programming language, we were faced with an application domain that was only poorly understood and for which there were few linguistic precedents. Software-process programs must be able to support software processes; a software-process programming language, in turn, must facilitate the coding of such programs. The ability to encode effective process programs can thus be taken as a general acceptance criterion for a process-programming language.[1] Consequently, we began by considering the general characteristics of software processes. These characteristics and other issues relating to the requirements on a process-programming language are discussed in Section 2.1. We next identified the kinds of capabilities that a software-process programming language would need to support the required characteristics of software processes. These capabilities, presented in Sec-

[1]In the following we will use "process programming" and "process-programming language" for "software-process programming" and "software-process programming language", respectively. Ideas about process and process programming apply to a wider domain than software development, but since we are concerned here specifically with software development, we will drop the qualification for the sake of brevity.

tion 2.2, became the design goals for our language. Finally, we adopted a strategy for defining a language to meet these goals as described in Section 2.3.

2.1 Requirements on Process-Programming Languages

A process-programming language is intended specifically for the coding of programs to represent and implement software processes. Consequently, there are domain-dependent requirements on these languages that derive from the particular needs of software processes. More specifically, a process-programming language should facilitate the development of process programs that capture and support the essential and desirable characteristics of software processes. Process-programming languages are also affected by domain-independent issues, such as principles of programming-language design, and the need to enable the development of process programs according to, and in support of, sound software engineering practices. However, the domain-specific requirements are of particular interest here because these have been the least-well understood of the issues affecting process-programming language design. Indeed, part of our purpose in developing a prototype process-programming language is to validate hypothesized language requirements based on the needs of software processes.

In order to determine language requirements that are derived from software processes, it is necessary to have some understanding of the particular characteristics of those processes that process programs are to address. The characteristics of software processes, in turn, depend on their purposes. We take those purposes to be the development, maintenance, evolution, and management of software products by teams of developers. Consequently, software development can be characterized not only in terms of "process" or behavioral concepts, such as methodologies and activities, but also in terms of products and of the interactions between processes and products. Previous work on software environments [10, 29] and early experience in process programming lead us to identify a number of characteristics related to each of these aspects of software processes. Some representative characteristics that seem especially important are summarized below. These, along with the more general issues noted above, motivate the language design goals presented in the following subsection.

Characteristics Related to Processes Software processes must have a *flexible and dynamic process structure*. For example, they must respond to contingencies that arise during product development, both exceptions and opportunities, and they must adapt to dynamically determined conditions such as the availability of resources and the size and complexity of the developing product. Software processes also typically include a *high degree of concurrency*. Many development activities may proceed concurrently, such as the coding of independent modules, and some classes of activity are necessarily parallel, such as an engineering activity and the managerial monitoring of that activity. Many development activities are also naturally characterized in terms of *reactive* (or *event-driven*) *control*. Examples include propagation of changes, the invocation of activities when needed inputs become available, and the repair of constraint violations or process failures.

Characteristics Related to Products Software products comprise *many kinds of artifacts*, ranging from requirements to design to different kinds of code, test cases, test plans, test results, analysis criteria and results, documentation, and so on. These artifacts furthermore have *diverse and complex interrelationships*. These include various kinds of dependencies (e.g., derivation and definition), versions, configurations, visibility, and so on. These relationships are critical not only for the comprehensive representation of software products but also for the effective representation of software processes, for example, in guiding development, assessing the impacts of change, and maintaining consistency. The *consistency* of software products is also important for a variety of reasons, such as the acceptability of a software product and pre- or postconditions on tool or process executions. Software processes thus need the ability to express and enforce consistency requirements on software products. Software products must also have *persistence*, that is, they must persist between (and following) the processes that develop, use, evolve, and maintain them. A related issue is *access to existing artifacts*, for example, for reuse or reengineering. These may be imported from external sources or inherited from prior projects. In either case, it implies a need to be able to incorporate the use of existing and possibly heterogeneous artifact stores into the software process.

Characteristics Related to Processes and Products or Their Interactions There are a number of areas in which characteristics of software processes apply to both processes and products or their interactions. One is a *substantial reliance on automation*. Notwithstanding the importance of manual activities, software development would not be possible without automated support for object creation, derivation, and manipulation (e.g., compilers and editors), tool and object management systems (e.g., for version control and configuration management), and other automated capabilities (such as communication and event notification). A need for *data sharing* arises because multiple programs often must use the same data and access them concurrently. For example, multiple analysis tools may be run on the same module concurrently, or a manager, design engineer, and programmer may need access to a design specification during a design review. Data sharing entails a *need to coordinate multiple users and processes* in order to resolve conflicting requirements or purposes and to mitigate contention for shared resources. The fact that different users or processes may be accessing shared data at different times and for different purposes implies a need for *flexibility in the specification and enforcement of consistency*. The conditions that represent consistency may evolve over the life of a process and product. For example, the constraints on a finished product may be much stronger than those on an initial or intermediate product. Different activities during development may require different levels of consistency (or, conversely, tolerate different amounts or kinds of inconsistency). For example, a tool for data flow analysis may require syntactic and semantic consistency of source code, but the semantic analyzer may require only syntactic consistency, whereas the use of a testing tool may be restricted to code that satisfies certain conditions on data flow. Finally, the different consistency requirements of different processes, and the fact that software development activities are prone to error, implies a need for the *ability to accommodate inconsistency.* Inconsistency may arise, for example, when constraints are newly enforced, when a change in one part of the process affects the consistency of another part, or when a tool or task fails.

2.2 Design Goals

The characteristics of software processes described above provide a basis for inferring domain-related design goals for process-programming languages. Based on these characteristics, and on general considerations of language design and software engineering, we adopted several specific goals for our prototype process-programming language. These are as follows:

- **Executability:** Executability is needed for process implementation in general to support the automation of process steps.
- **Basic control and data definition capabilities:** These are needed for representing aspects of software-process activities and artifacts; they are also important for programming the process-independent details of process implementations.
- **Abstraction, encapsulation, and modularity:** Abstraction, encapsulation, and modularity of both control and data are important for software-engineering purposes such as understandability, maintainability, and reusability. They provide a basis for software-process evolution and adaptation by facilitating the introduction of alternative methods, policies, and representations.
- **Representation of relationships:** This is important for capturing the diverse interconnections between software products and also for representing other associations in a software process and environment (e.g., the relationship of resources to activities).
- **Derived data:** The capabilities for automation and abstraction should be sufficient, as a special case, to support the specification, implementation, and maintenance of derived data. Derived data are important because many essential kinds of artifacts (e.g., object and executable code, test and analysis results) seem to be effectively represented as derived data. Also, derivations represent an important opportunity for process automation.
- **Data persistence and sharing:** Persistence and sharing are needed for software products as well as for information that represents software processes, projects, and so on.

- **Programmability of implementations:** This complements control and data abstraction in the provision of alternative methods, policies, and implementations. As a particular case, it should be possible to program the representations of persistent data, for example, with regard to the underlying storage system, storage strategy, inferencing, and so on. This facilitates logical independence of products and processes from underlying systems while enabling optimization of the implementations and supporting access to legacy data.

- **Concurrent and reactive control:** These address the need for concurrent and reactive activities in software processes. Both may also contribute to flexibility and dynamism of software-process control structures.

- **Explicit, flexible, and dynamic consistency management:** This addresses the need for expressed and enforcible constraints on data as well as for control over the enforcement of those constraints. Flexibility and dynamism in consistency enforcement can also contribute to flexibility and dynamism in software-process control structures and to the accommodation of inconsistency.

- **Flexible transaction management:** Transaction-related capabilities are needed to help coordinate concurrency among users and processes in their access to shared and persistent data, in particular to prevent interference between concurrent activities and preserve data integrity. Flexibility of the transaction model is needed to help support process-specific transaction models. It can also contribute to flexibility and dynamism in software process control structures and to the accommodation of inconsistency.

These goals represent general capabilities that we hoped to support in some specific form in our prototype process-programming language. Two things should be noted about the goals. First, several of them have multiple motivations. For example, modularity and abstraction are important in language design and software engineering generally, but they have particular relevance for software processes in that they facilitate the adaptation and evolution of methodological policies and environmental infrastructure. Second, language capabilities can be categorized analogously to the characteristics of software-process, that is, in terms of control and data and the interactions between them. However, it would be incorrect to infer a fixed association between the behavioral aspects of processes and the control-related elements of languages, or between the product-related aspects of processes and the data-related aspects of languages. In other words, a given language feature cannot be classified simply as a "process feature" or a "product feature"; the features are mechanisms that can be applied to multiple purposes. In practice, both the control-related and data-related capabilities of a language are useful for programming both the behavioral and product-related aspects of software processes (as illustrated in Section 5 and discussed in Section 6.10).

We recognize that there are issues that are not directly addressed by the design goals stated above. Rather than attempt to address all conceivable process-programming language requirements, we deliberately narrowed our focus to concentrate on aspects of software processes and products that seem important and interesting to us. Some issues, such as user interfaces and real-time control, are beyond the scope of our work. Capabilities such as high-level transactions and version control are necessary to software processes, but we could identify no generally accepted models for these that we felt comfortable embedding in the language. In these areas we have tried to supply more basic constructs that can be used to program needed capabilities according to process-specific requirements. Other issues, such as user-process interaction paradigms and process change, we planned to explore as issues of process-program design rather than language design. Finally, there are some issues that we did not consider in the original design of APPL/A but that have since been recognized as important. These include concerns such as persistent (or long-term) and reflexive processes. For many of these "new" problems we have found that capabilities already in the language provide reasonable solutions (see Section 6).

2.3 Development Strategy

Having identified design goals for our prototype process-programming language, the next step was to define the language according to those goals. However, it was not possible *a priori* to specify the particular form in which the language should provide the desired capabilities. As noted, there were few demonstrated control and data models for software processes. Additionally, we knew of no programming language, intended for any purpose, that provided support in all of the required areas. Consequently, we expected to have to modify existing models and define new ones to provide the desired capabilities. Furthermore, we could not know in advance how potential control and data models might interact; thus, we were faced with the prospect that the form and function of features in the language might need to be adapted and restricted if they were to be coherently integrated. The definition and integration of control and data models was thus a substantial part of our effort.

We began the definition of our language with the premise that, since process programs are intended to serve as process implementations, a process-programming language should provide at least the kinds of features and capabilities offered by traditional programming languages. Traditional programming languages could address at least some of the goals we have for process programs, such as executability and basic control and data definition. Many of the goals reflect fundamental language-design principles and are intended to simply facilitate sound software-engineering practices. Traditional languages may also enable the representation and support of some of the characteristics of software processes (e.g., concurrent behavior), although we expect that in general capabilities beyond those in traditional programming languages will be needed.

On the assumption that a process-programming language should largely subsume a traditional programming language, we decided to define our prototype process-programming language by extending an existing language with features that we believed would be important for software-process programming. We chose to extend an existing language so that we would not have to reinvent the basic capabilities provided by superior contemporary languages and because we hoped that this would simplify implementation of the process-programming language and ease its integration into existing software-development contexts. This would also enable us to more quickly gain experience with the language and with process programming and it would expedite the evaluation of our approach.

We selected Ada for our base language. We hypothesized that the capabilities of Ada that were intended to facilitate software engineering would also be useful in a process-programming language. These capabilities include abstraction, modularity, concurrency, and exceptions and exception handlers, among others. It also seemed that some of the particular constructs in Ada, such as packages and tasks, could serve as syntactic and semantic models for the new features to be added. Additionally, we hoped to take advantage of Ada-related technology that was being developed in the Arcadia project [71], through which the work on APPL/A was supported. An overview of the special features of APPL/A, defined as extensions to Ada, is given in the next section.

3 Overview of APPL/A

This section presents an overview of the definition of APPL/A. APPL/A is defined as an extension to Ada (and we assume below that the reader is at least somewhat familiar with Ada). The major constructs that APPL/A adds to Ada are

- Relation units that provide abstract, programmable, persistent storage for writable data
- Trigger units that represent concurrent logical processes, like Ada tasks, and that react automatically to operations on relations
- Predicate units that specify conditions over relations and that can be dynamically and optionally enforced to achieve the effect of constraints on relations
- Composite statements that provide capabilities for management of relation integrity and consistency and that can be used to program activities with transaction-related properties such as serializability and atomicity.

APPL/A also defines a variety of other statements and operations related to these. The full definition, including additional features and details, is presented in [62]. The subsections below present an overview of the main points relating to the most important and widely used constructs in the language.

3.1 Relation Units

APPL/A relation units are used to represent the abstract mathematical notion of a relation (except that APPL/A relations are by default multi-sets). Relations address the goal of representing relationships among software objects and products, and they also provide a general and flexible data structure for representing software products and other sorts of software-process data. Like relations in conventional databases, APPL/A relations are persistent and shared between programs. Thus relations also address our goal of providing support for shared, persistent data.

APPL/A relations have several important differences, however, from relations in the conventional relational data model [11]. The types of attributes in APPL/A relations can be composite and abstract, although they must be assignable by value. APPL/A attribute values can be automatically computed, providing support for our goal of derived data. APPL/A relations also have programmable implementations, which allows implementations to be tailored to an environment while preserving environment-independence at the abstract level. Another unusual feature of APPL/A relations is that they are "active", i.e., they represent concurrent threads of control. This contributes to the goal of concurrency, especially for purposes such as attribute derivations, inferencing, and dynamic storage management.

Syntactically, an APPL/A relation declaration consists of a specification and a body, analogously to Ada package and task units. Also like Ada tasks, a relation specification may define a single instance of an anonymous type, or it may define a type which can be multiply instantiated.

3.1.1 Relation Specifications

An APPL/A relation specification contains a tuple type definition and one or more entry specifications (taken from a pre-defined set, described below). If the relation has derived attributes, then it may optionally include a *dependency specification* to specify how those attributes are to be computed. These features are illustrated in Figure 1. This shows the specification of a relation `Source_Compilations`, which relates object code to the source code from which it is compiled.

The tuple type definition in a relation specification defines the type of tuples that are stored in the relation. A tuple type is similar to a record type, but the fields of a tuple are known as "attributes."[2] Attributes, like Ada parameters, have modes (**in**, **in out**, and **out**) to indicate the way that the attributes take on values (described below).

The entries for a relation represent the operations on the relation. These must be at least one of

[2]The attributes of the tuple type of a relation are also called the attributes of the relation. This term was chosen for conformance with established relational terminology. "Attribute" is also used in Ada to refer to predefined properties of named entities. We regret this overloading of "attribute", but we felt that abandoning the use of the term with respect to relations would create even more confusion.

insert, update, delete, and find. The insert entry takes parameters for **in** and **in out** attributes and inserts a tuple with the given values into the relation. The update entry enables one tuple with given attribute values to be assigned new values for the **in** and **in out** attributes. The delete entry deletes a tuple with given attribute values. The find entry supports the iterative and selective retrieval of tuples according to given attribute values.[3] Relation entries, like those for a task, are executed serially; relation entries must also be atomic, i.e., have an all-or-nothing effect on the state of the relation. (These properties are necessary to assure the proper behavior of relations with respect to predicate enforcement, Section 3.3.2, and the consistency-management statements, Section 3.4.)

Two other operations on relations are not represented by entries but are predefined by the language. These include a function member, which tests whether any tuple in the relation matches given attribute values, and a function tuple, which returns a tuple matching given attribute values (or raises an exception if there is no such tuple).

Various effects can be achieved by omitting one or more of the relation entries. For example, a "constant", read-only relation can be defined by omitting the insert, update, and delete entries. A write-once/read-many relation could be defined by omitting the delete and update entries. Such a relation might be used, for example, for process logging. Other properties related to relation operations are discussed in Section 3.4.

A dependency specification indicates how the values of derived attributes (modes **out** or **in out**) are to be computed. In Source_Compilations the dependency specification states that, for each tuple, the values of the attributes obj and msgs are to be computed by a call to the procedure compile, where the corresponding value of attribute src is taken as input.

3.1.2 Relation Bodies

The body of a relation must implement the semantics of the associated specification. That is, it must provide persistent storage, implement the relation entries, and compute and assign values for derived attributes. However, many details of the implementation are left up to the programmer (although a default implementation mechanism is available). In this respect APPL/A relations are *programmable*. Thus, for example, the implementation of a relation is *not* constrained with respect to the system used for persistent storage, the derivation strategy for computed attributes (e.g., eager or lazy), the caching strategy for computed attributes (e.g., cached when computed or recomputed when needed), or any inferencing that may be necessary to support the implementation. This aspect of APPL/A, which is based on [29], goes beyond earlier systems such as Odin [10] and Make [19], which were less flexible or powerful in these

[3] The find entry is used to implement a relation iterator (loop construct) and the predefined relation operations member and tuple. Typically the user does not call the find entry directly but uses these other constructs to retrieve or test for tuples.

```
with Compile; -- separately defined compiler interface
with Code_Types;
--
Relation Source_Compilations is
-- Relates source code to the object code compiled from it.
-- Encapsulates and automates the compilation process.
--
    type src_compilations_tuple is tuple
        name: in name_type;
        src: in Code_Types.source_code;
        obj: out Code_Types.object_code;
        msgs: out messages;
    end tuple;
entries
    insert(name: name_type; src: source_code);
    delete(name: name_type; src: source_code; obj: object_code; msgs: messages);
    update(name: name_type; src: source_code; obj: object_code; msgs: messages;
        update_name: boolean; new_name: name_type;
        update_src: boolean; new_src: source_code);
    find(iterator: in out integer;
        first: boolean; found: out boolean;
        t: out src_compilations_tuple;
        select_name: boolean; name: name_type;
        select_src: boolean; src: source_code;
        select_obj: boolean; obj: object_code;
        select_msgs: boolean; msgs: messages);
dependencies
    determine obj, msgs by compile(src, obj, msgs);
End Source_Compilations;
```

Figure 1: Specification for Relation `Source_Compilations`

respects. Relations are thus abstract data types; as such, they afford the benefits generally associated with abstraction, for example, encapsulation and the flexibility to change implementations without affecting users of the specification.

3.1.3 Persistence and Sharing of APPL/A Relations

The data in an APPL/A relation by definition must persist between program executions. This applies not only to relations that are library units but also to relations that are declared in local scopes (so long as they are reachable).[4] The scoping and visibility rules for APPL/A relations are the same as

[4]Relations can be designated by access values. Since access values are meaningful only within a single program execution, such relations need not persist between executions.

those for Ada objects. There is only one relation object for each declared relation, including relations declared in local scopes. Concurrent executions of programs (or scopes) in which a relation is visible give rise to concurrent references to the single object represented by the declaration. Mechanisms that allow programs to coordinate concurrent access to shared relations are provided by the transaction-like consistency-management statements described in Section 3.4.

3.1.4 Iteration Over Relations

APPL/A provides a "for" loop for (optionally selective) iteration over readable relations (i.e., relations that have a `find` entry). For selective iteration, a **where** clause is used to specify one value each for one or more attributes of the relation; only tuples that match the given value(s) are retrieved. For non-selective iteration, the **where** clause is omitted. An example is shown within Figure 2.

3.2 Trigger Units

An APPL/A trigger unit represents a separate logical thread of control that *reacts* to events defined in terms of calls on relation entries. Triggers thus address our goals for both reactive and concurrent control in software processes. Triggers can be used to automatically propagate updates from one relation to another, to send notifications in response to changes in data that are stored in relations, to initiate dependent processes, to perform computations, to maintain logs, and generally to respond concurrently to operations on relations.

3.2.1 Events, Event Signals, and Trigger Execution

For purposes of triggering, APPL/A defines two kinds of *events* in association with relation entry calls. An *acceptance* event occurs when a relation entry call is accepted; a *completion* event occurs when a relation entry call is successfully completed. When an event occurs, an *event signal* is generated and automatically propagated to triggers that respond to that event. An event signal carries with it the values of actual parameters given to or returned from the corresponding entry call, and these are made available to triggers that receive the signal.

The execution of triggers is modeled after that of Ada tasks in that a trigger logically executes in parallel in the way that a task does. A difference between triggers and tasks, though, is that a task responds to explicit entry calls, while a trigger responds to implicitly generated event signals. More particularly, a typical trigger "waits" for event signals that are generated automatically in association with operations on relations. When an event signal is received, the trigger responds by executing the code associated with the event (represented by an **upon** statement in the trigger body, as discussed below); the trigger then resumes "listening" for further event signals. If signals are received by a trigger more rapidly than they can be handled, they are queued (like calls to a task entry) and responded to in turn.

Note that the association of a trigger to a relation is represented in the trigger only; i.e., it is not necessary in a relation to designate the triggers to which event signals are to be sent. Thus, triggers on a relation can be added and deleted without affecting the relation; in this regard, the APPL/A event and trigger model is consistent with the event/mediator model of environment integration advocated in [61].

3.2.2 Trigger Specifications

A trigger has a one-line specification like that of an Ada task with no entries. A trigger specification with the reserved word **type** defines a trigger type; one without the reserved word **type** defines an instance of an anonymous type. A trigger may also be declared **global** or **local**. This determines the scope from which event signals are propagated to the trigger (global is the default).

The specification and body of a trigger are shown in Figure 2. The purpose of this trigger, `Maintain_Source_Compilations`, is to automatically assure that every module in relation `Source_Repository` (not shown) is represented in relation `Source_Compilations`. `Source_Repository` has attributes **author** (module author), **name** (module name), and **src** (source code). The trigger responds to operations on `Source_Repository` and propagates corresponding changes to `Source_Compilations`. For example, when new source code is inserted into `Source_Repository`, the trigger automatically inserts that code into `Source_Compilations`.

3.2.3 Trigger Bodies, Upon Statements, and Synchronization

A trigger body comprises a loop over a selective trigger statement in which the alternatives are represented by **upon** statements. The **upon** statements identify the events to which a response is to be made. Each **upon** statement designates a relation entry and takes corresponding parameters. The keyword **acceptance** or **completion** designates the kind of event associated with the entry. The statements within and immediately following the **upon** statement encode the trigger's response to the event. A trigger may have **upon** statements for any or all of the entries of a relation, and it may have **upon** statements for entries from multiple relations.

The body of an **upon** statement (within the **do** ... **end** block) is executed synchronously with the event signal. While the **upon** statement is executing, the execution of the corresponding relation is suspended at the point at which the signal was generated. However, the trigger does not execute a full rendezvous (in the Ada tasking sense) with the relation, and no parameters or exceptions are returned from the trigger to the relation. Once the **upon** statement completes, the relation is released and the trigger and relation

```
global trigger Maintain_Source_Compilations; -- the specification

trigger body Maintain_Source_Compilations is
    sc_t: src_compilations_tuple;
begin
    loop
        select
            upon Source_Repository.insert(
                author: name_type; name: name_type; src: source_code)
            completion do
                -- propagate name and source to Source_Compilations
                Source_Compilations.insert(name, src);
            end upon;
        or
            upon Source_Repository.update(
                author: name_type; name: name_type; src: source_code;
                update_author: boolean; new_author: name_type;
                update_name: boolean; new_name: name_type;
                update_src: boolean; new_src: source_code)
            completion do
                if update_name or update_src then
                        ... -- update tuple with "name" in Source_Compilations
                end if;
            end upon;
        or
            upon Source_Repository.delete(
                author: name_type; name: name_type; src: source_code)
            completion do
                -- delete tuple with "name" from Source_Compilations
                for sc_t in Source_Compilations where sc_t.name = name loop
                    Source_Compilations.delete(sc_t.name, sc_t.src, sc_t.obj, sc_msgs);
                end loop;
            end upon;
        or
            terminate;
        end select;
    end loop;
End Maintain_Source_Compilations;
```

Figure 2: Trigger `Maintain_Source_Compilations`

proceed in parallel and asynchronously.

3.2.4 Global and Local Triggers

Global triggers receive signals from all events of the types designated in their **upon** statements, regardless of the location in a program of the entry call that generated the event. Local triggers receive signals from only those events that are generated by entry calls within the innermost scope that contains the trigger. Local triggers can thus be used to program "process-specific" triggering. For example, a local trigger can be used to log or send notifications for relation operations performed within a particular task, without having to filter out events related to operations performed by other tasks. Global triggers provide a complementary mechanism for the programming of "process-independent" triggering in that these triggers can track events arising from any program in the environment.

3.3 Predicate Units and Consistency

APPL/A predicate units and their related mechanisms represent one part of a two-part approach toward our design goals of explicit and flexible consistency management, including the accommodation of inconsistency. The second part is the consistency-management statements, which are presented in Section 3.4. A discussion of how predicates and consistency-management statements may be used together to manage consistency and accommodate inconsistency is found in Section 3.5.

An APPL/A predicate unit allows the specification of a condition on relations. Predicates can be explicitly invoked, like functions. Predicates can also be automatically enforced, like constraints, and the choice of whether or not to enforce a predicate can be made (and changed) dynamically. Predicates can thus be used to define and enforce consistent states for relations, and the ability to change the predicate enforcement allows the criteria for consistency to change dynamically. Because predicates apply to relations, and because relations are widely applicable in the representation of both software products and processes, predicates are likewise widely applicable in specifying and managing the consistency of software products and processes.

3.3.1 Predicate Syntax

A predicate unit is a named boolean expression over designated relations. The expression language includes existentially and universally quantified forms and conditional expressions. Three predicates are shown in Figure 3; these refer to a relation `Source_Repository` (described above) that stores source modules with the module name and author. The first predicate simply tests a property of the `src` (source code) attribute of `Source_Repository`, namely, that all source modules have a length not more than `Max_Lines`.

```
-- a predicate on a property of an attribute
predicate Source_Length_Within_Limits is
begin return
    every t in Source_Repository satisfies
        Length(t1.src) <= Max_Length
    end every;
End Source_Length_Within_Limits;

-- a predicate on referential integrity
mandatory predicate Compiled_Modules_in_Repository is
begin return
    every t1 in Source_Compilations satisfies
        some t2 in Source_Repository satisfies
            t1.name = t2.name
        end some
    end every;
End Compiled_Modules_in_Repository;

-- a predicate on uniqueness of names
mandatory enforced predicate Name_Unique_in_Source_Repository is
begin return
    every t1 in Source_Repository satisfies
        no t2 in Source_Repository satisfies
            t1.name = t2.name
        end no
    end every;
End Name_Unique_in_Source_Repository;
```

Figure 3: Two APPL/A Predicates

The next predicate tests the integrity of name references between relations `Source_Compilations` and `Source_Repository`. The third tests the uniqueness of names in `Source_Repository`. Keywords used in these examples are explained below.

Predicates that are declared with the reserved word **mandatory** define conditions (and potential constraints) that are global with respect to the relations to which they apply. They must be visible wherever the relations to which they refer are visible. (This creates, in effect, an implicit compilation dependency between those units and the predicates.) A predicate that is not declared **mandatory** is called "optional." Optional predicates may be included along with the relations to which they apply but are not required. Optional predicates thus define conditions (and potential constraints) that may be only locally applicable within the scope of referenced relations. A **mandatory** predicate may also be declared **enforced**, which means it is always enforced by default, as explained below. Whether a predicate is mandatory or optional affects not just its visibility but also the scope over which it is enforced.

3.3.2 Enforcement

An enforced predicate acts like a post-condition on relation operations. When a predicate is enforced during the execution of a program, no operation by that program on relations to which the predicate refers may terminate in violation of the predicate. Any operation resulting in a violation is undone and causes an exception to be raised. In this way an enforced predicate acts something like a constraint on the relations to which it applies.

A predicate may or may not be enforced by default. For mandatory predicates the default enforcement is initially "on", while for optional predicates it is initially "off." A predicate declared **enforced** is always enforced by default; the default enforcement of other predicates can be set (and modified) dynamically. (The consistency management statements, described below, provide a way to locally override the default enforcement of any predicate.)

Each mandatory predicate has one default enforcement for all programs in which it occurs; i.e., it is enforced by default in either all or none of those programs. In contrast, each optional predicate has a separate default enforcement in each program in which it appears; i.e., the predicate may be enforced in one program but not another. Note that the default enforcement of a predicate may be turned on in a state in which the predicate is not satisfied. This simply results in an inconsistency, which may be handled in any of the ways described in Section 3.5.

The above rules provide flexibility with respect to when and where consistency conditions are enforced. For example, consider a graph represented by two relations, one for nodes and their attributes, the other for edges (designated by pairs of nodes). Such a graph may be used to represent various software products, such as requirements specifications of the type supported by **REBUS** (Section 5.1). A mandatory enforced predicate might be used to assure the referential integrity between the edge and node relations, i.e., that every edge refers to nodes that occur in the node relation. This predicate would be, in effect, a global constraint. Its enforcement could not be turned off, and it could be violated only locally and temporarily within a transaction-like **suspend** or **allow** statement (see Section 3.4). This is appropriate for a condition that reflects the fundamental semantic integrity of the graph, one that may be required, for example, both by processes that construct the graph and by processes that may use it (e.g., analysis tools). A mandatory (but not enforced) predicate might express the condition that the graph is connected. As a mandatory predicate, this could be enforced as a global constraint, but the enforcement could be turned on and off over time according to the phase of development and the role or state of the graph. Finally, an optional predicate might be used to express the condition that the graph is acyclic. Depending on the semantics of the graph, this condition may be important only to certain analysis tools. The predicate may be enforced within these tools but need not be imposed elsewhere. This allows other tools (such as graph-building tools) to operate free from the constraint (and from the need to check it), but it does allow those tools to violate the condition, which would create an inconsistency for the analysis tools. Thus, such predicates should only be used when there is a reasonable expectation that the condition will not be violated or that the exception that results from a violation can be handled.[5]

3.4 Transactions and Consistency-Management Statements

The consistency management statements in APPL/A are intended to provide the transaction-related capabilities that are needed in software-process programs. Consequently, they are defined in terms of the basic properties usually associated with transactions. These are serializability, atomicity, and consistency. Two concurrent transactions are *serializable* if their combined result could have been obtained by executing them in some serial order. An *atomic* transaction has an all-or-nothing effect; it produces a complete result if it succeeds and no result if it fails. Finally, a *consistent* transaction is one that satisfies all implicit or explicit constraints on data.

APPL/A relation entry calls individually have properties like those of a one-step transaction. They are serial (because a relation may accept at most one call at a time), they must have an all-or-nothing effect on the state of the relation (which must be enforced by the relation implementation), and each must satisfy the predicates that are enforced on the relation (or the operation is undone).

APPL/A consistency-management statements support the programming of multi-step, transaction-like operations on relations. They are intended to support activities where concurrency control, atomicity, and consistency are important, for example, in a multi-developer requirements-specification process, or in coordinating design updates with concurrent coding of modules that are based on the designs. The statements provide various combinations of serializability, atomicity, and control over predicate enforcement. Serializability assures that concurrent consistency-management statements do not interfere with one another through their operations on shared relations. The APPL/A model of serializability presumes the locking of relations for reading or writing. Read locking allows concurrent access, while write locking provides exclusive access (thus imposing serial execution of the statements on write-locked relations). Atomicity implies a need for the ability to negate the (possibly incomplete) effects of a failed consistency-management statement. The APPL/A model of atomicity is based on logging, with undo in the event of failure. A consistency-management statement that is atomic in this sense is called *recoverable*.[6]

[5]Note that if it is truly important to protect the analysis tools from the violation of some condition then a mandatory enforced predicate should be used instead of an optional predicate.

[6]Because the serializability and atomicity of consistency-management statements apply only to operations on relations, it is possible for a consistency-management statement to exhibit non-serializable or non-atomic behavior with respect to other operations. While this is typical of many database programming languages, it must be recognized that a consistency-management statement that is to be strictly serializable or atomic should not operate on other data or perform I/O.

In order to address our design goal of a flexible transaction model, the consistency-management statements are more flexible and general than conventional database transactions. A typical database transaction combines serializability, atomicity, and the temporary relaxation of constraints. The consistency-management statements instead provide various alternative combinations of these capabilities; the statements also offer a wider range of capabilities, for example, the ability not just to relax constraints but also to impose them. Thus the statements make it possible for a program to more precisely match the needs of a process where some but not all of the capabilities are required.

The APPL/A consistency-management statements can also be composed to create more elaborate transaction-related control structures. The statements can be nested (based on the nested transaction model of Moss [47]), and they can include concurrent tasks or triggers and subroutine calls; in effect, they may include any arbitrary APPL/A program. In this way the statements support the programming of process-specific "high-level" or "long-term" transactions, such as long-duration saga-like operations [22], workspaces with check-in and check-out, and cooperative-work applications (see Section 5.3). In combination with mechanisms associated with APPL/A predicates, they provide a flexible approach to managing consistency and accommodating inconsistency.

3.4.1 The Consistency-Management Statements

The APPL/A consistency-management statements are the **serial**, **atomic**, **suspend**, **enforce**, and **allow** statements. They are modeled after Ada block statements. Each statement also has additional syntax to designate either the relations to which it requires read or write access or the predicates for which it is locally suspending or imposing enforcement. The properties of the statements are summarized in Table 1. Brief descriptions and examples of the statements follow.

Statement	**Serializable**	**Atomic**	**Enforcement-Setting**
Serial	yes	no	no
Atomic	yes	yes	no
Suspend	yes	yes	suspends
Enforce	no	no	enforces
Allow	yes	yes	suspends

Table 1: Summary of Properties of Consistency-Management Statements

Serial Statement The serial statement provides serializable read or write access to relations designated in a "read-write" list. It does not provide atomicity and it does not affect predicate enforcement. The serial statement thus enables concurrent processes to coordinate their access to relations without imposing the run-time burden of atomicity and while maintaining the enforcement of relevant predicates. An example is shown in Figure 4. During the execution of this statement, other users can read relation `Source_Repository`, but no concurrent process can access `Source_Compilations`. No special access is requested for relation `Error_Log` (not shown), but it can be written nevertheless, so long as that writing does not conflict with the access held by any other operation or statement.

```
copy_source:
serial read Source_Repository; write Source_Compilations;
begin
          ... -- copy source modules from Source_Repository into Source_Compilations
exception
     ...; -- record problem in relation Error_Log
end serial copy_source;
```

Figure 4: An APPL/A Serial Statement

Atomic Statement The atomic statement provides both serializable and recoverable access to relations designated in a "read-write" list. Write operations on the designated relations are implicitly logged, and the propagation of an exception from the atomic statement causes rollback of those operations and propagation of the exception. The atomic statement thus provides a unit of work that is atomic with respect to *completeness* (where an exception is taken to signal a fatal incompleteness). The atomic statement does not affect predicate enforcement, so consistency is maintained within the statement. The atomic statement is serializable to prevent outside access to intermediate results that may be undone if the statement fails.

Suspend Statement The suspend statement provides a context in which the actual enforcement of designated predicates is temporarily and locally suspended. This allows composite updates to relations where the intermediate results may not preserve consistency but the final result does. The suspend statement implicitly provides serializable and recoverable write access to the relations to which the suspended predicates apply. Write operations on these relations are logged. Upon termination of the body of a suspend statement, all of the suspended predicates that are enforced in the surrounding scope must be satisfied; otherwise, the logged operations are automatically rolled back and an exception is raised. The suspend statement thus provides a unit of work that is all-or-nothing with respect to *consistency* (defined in terms of enforced predicates). The statement is serializable because its intermediate results are subject to undoing and may also be inconsistent. A suspend statement is shown in Figure 5.

```
... -- Predicate Source_Length_Within_Limits is enforced here
edit_source_modules:
suspend Source_Length_Within_Limit;
begin
    ... -- Edit modules in Source_Repository. Source_Length_Within_Limit
        -- is not enforced here; other predicates are enforced
end suspend edit_source_modules;
-- Source_Length_Within_Limit must be satisfied at end of suspend
-- or operations on Source_Repository are undone
```

Figure 5: An APPL/A Suspend Statement

Enforce Statement The enforce statement provides a context in which the actual enforcement of designated predicates is temporarily and locally *imposed* rather than suspended. It can be used to strengthen consistency requirements locally without affecting the consistency requirements of other processes. Any operation within the scope of the enforce statement that violates an enforced predicate is individually undone as it occurs. Consequently, there is no rollback for the statement as a whole. Because the enforce statement does not introduce the possibility of rollback, and because it does not weaken the consistency of data, the statement is not serializable.

Allow Statement The allow statement, like the suspend statement, creates a context in which the enforcement of predicates is suspended. However, designated predicates that are violated upon entry to the allow statement may still be violated when the statement terminates. In other words, the allow statement allows an *existing* predicate violation to be perpetuated. The purpose of this is to enable the partial repair of predicates that are already violated while preventing the violation of additional predicates. Any operation that violates an enforced predicate that is not suspended is individually undone (with the raising of an exception); thus consistency with respect to these predicates is preserved within the statement. There is no rollback for the statement as a whole since that would preclude the possibility of partial repair. However, because the statement suspends the enforcement of violated predicates, it is serialized with respect to operations on the relations referenced by those predicates.

3.4.2 "Detached" Consistency-Management Statements

Most of the consistency-management statements preclude outside processes from accessing their intermediate results; for the recoverable statements, these results will never be available if the statement fails. While it is often important to exclude outside access to intermediate (and possibly transitory) results, there are circumstances under which it may be useful to provide access to such results on a selective basis. For example, a design process may be treated as a long transaction from which intermediate results are exported to facilitate early prototyping by a coding process. Even if the overall design process fails, some of the designs produced may be reusable in later iterations of the process. Similarly, a process may record information about its execution history, control decisions, and so on. This information may most useful specifically if the process fails, for example, in analyzing, debugging, and redoing the process. To address this need, APPL/A includes the idea of a "detached" transaction.[7] In APPL/A, any consistency-management statement or relation operation may be preceded by the keyword **detached.** A detached statement or operation is serializable and recoverable independently of any enclosing statement or operation. In these respects, it is as if the detached statement or operation were coded outside of the enclosing statement or operation. However, with respect to visibility, the usual rules associated with lexical nesting still apply. Thus a detached statement or operation can read data that are declared in an enclosing statement or operation, but the failure of the enclosing statement or operation does not imply the undoing of the detached statement as it would for ordinary nested statements or operations.

3.5 Managing Consistency and Accommodating Inconsistency

In accordance with our language design goals, the APPL/A consistency model is intended to support a rigorous yet flexible approach to the management of consistency. It is motivated by two ideas: that the criteria for consistency are evolving and relative, and that inconsistency must be accommodated as a normal condition.

The idea that criteria for consistency evolve and are relative is based on the recognition that different phases or activities in software development may require or support different kinds or degrees of consistency. For example, consider a requirements-specification process (like **REBUS** [67]) that uses a directed, acyclic graph to represent the functional requirements for a product. At first the requirements may be specified piecemeal, with different developers providing different subgraphs corresponding to different areas of the functional requirements; these initial subgraphs need not be connected or even acyclic. Later, the subgraphs may need to be connected to support analysis for cycles. After that, the graph may need to be connected and acyclic to support semantic analyses of the requirements. Finally, the graph may need to be connected, acyclic, and semantically consistent before it is released for use in developing a design. If problems in developing the design lead to refinements of the requirements specification, then the consistency of the requirements may subsequently need to be relaxed and reestablished. In these sorts of ways and others the criteria for consistency may depend on the phase of development or the particular activity being performed.

The idea that inconsistency should be accommodated as natural is based on the recognition that it is practically inescapable and that it may arise as a result of deliberate, unexceptional behavior [64, 4].

[7]"Detached transactions" were originally called "separate transactions", which seems a better term, but "separate" was already used as a keyword in Ada.

Inconsistency may arise for many reasons. Errors and failures may lead to inconsistency, e.g., when a repository is corrupted or a tool crashes (or just operates incorrectly). In these cases, it may be appropriate and adequate to treat resulting inconsistency as an exception. However, inconsistency may arise even when tools and systems operate correctly. When the criteria for consistency evolve, new conditions for consistency may be violated by existing data. When concurrent activities require or support different conditions for consistency, conflicts between them may also lead to inconsistencies. It has been pointed out that inconsistency may also result when constraints must be satisfied by the combined actions of independent, asynchronous tasks [4].

Rigor in the APPL/A consistency model is supported by the ability to explicitly state and enforce the conditions that define consistency. Rigor is also supported by the serializability and atomicity of the consistency-management statements, which assure that inconsistent or incomplete results are not available to outside processes. Flexibility in consistency management is provided through several mechanisms. These include the ability to define predicates with global or local scope and with mandatory or optional enforcement.[8] Predicate enforcement can also be controlled dynamically through the **enforced** attribute, and the actual enforcement of any predicate may be locally and temporarily suspended or imposed using a consistency-management statement.

The APPL/A consistency model accommodates inconsistency by allowing it to occur as a consequence of the correct and normal behavior of programs and by admitting a variety of responses to inconsistency. Inconsistency may occur when a predicate is newly enforced under conditions in which it is not satisfied. Inconsistency may also occur when a predicate that is optionally enforced in one process is violated by another process (in which the predicate is not enforced). This reflects a situation in which concurrent processes have conflicting criteria for consistency.

The features that provide flexibility in consistency management also allow inconsistency to be tolerated and repaired. The enforcement of a violated predicate may be turned off (if the predicate is not mandatory), thus reestablishing consistency by weakening the consistency criteria. Alternatively, the data that violate an enforced predicate may be updated so that the predicate is satisfied, thus reestablishing consistency by "strengthening" the state of the data. A **suspend** or **allow** statement can be used to make complete or partial repairs, respectively, to data. When an optional predicate is violated, consistency may also be restored by a separate process in which the predicate is not enforced.

[8]Note that, for purposes of enforcement, a predicate declared in an outer scope cannot be hidden by a predicate declared in an inner scope.

4 Status of the Language

A complete definition for APPL/A is found in [62]. It is presented in the style of the Ada manual, with a formal grammar, semantics defined in English, and small examples. Since this definition was completed we have identified many refinements that may be made in a subsequent version of the language.

An automatic translator, APT [26], is available for a subset of APPL/A. APT makes use of Arcadia language-processing technology. It works by preprocessing APPL/A code, supplemented with additional syntactic markers, into the internal representation for an equivalent Ada program. (This internal representation is an IRIS graph [3].) The internal representation is then converted to Ada source text, which can be compiled by standard Ada compilers. APT will translate specifications and bodies for relation and trigger units, including related constructs such as relation iterators, relation entry calls, and upon statements. The run-time system supports the full functionality of these units, including event signaling between separately executing programs. The implementation of the event signaling mechanism is based on Q [44], and is coded in part in APPL/A. APT will also translate predicate units, but automatic enforcement of predicates is not yet supported. APT will also parse the consistency-management statements, but they are not yet implemented. A detailed design has been developed, in APPL/A, for the runtime system necessary to support the functionality of predicate enforcement and the consistency-management statements, and plans have been made to extend the APPL/A implementation in these areas.

TARGeT is a tool for the automatic generation of relation bodies based on the Triton [28] object manager. However, APPL/A programs have made use of a variety of systems to provide the persistent storage required for relations. These systems include Pleiades [70], Cactis [30], and Ada direct I/O files. The language is designed so that no specific object-management system is required, and we have taken advantage of this flexibility in our programs.

5 Experience

We have made substantial use of APPL/A in programming software processes and other applications. This experience includes both "paper" studies and fully implemented, executable systems. The paper studies have been conducted in cases where we wished to test the expressivity of the full language (including the parts for which we do not yet have automated translation) or in cases where language or design issues could be explored without programming complete implementations. Examples of these kinds of studies include the APPL/A "solutions" to the process problems for the Sixth and Seventh International Software Process Workshops (described in Sections 5.2 and 5.3). The fully implemented systems represent process programs and tools that provide needed functionality or demonstrate results that can be achieved through operational processes and process programs. Examples of these sorts of systems include process programs and tools in the Arcadia demonstration environment and other experimental prototypes (described in Sections 5.1, 5.4, and 5.5).

Our use of APPL/A has enabled us to validate many language features, to identify limitations in others, to explore process-program design issues, and to investigate software processes. The lessons we have learned from this experience are summarized in the next section (Section 6). In this section we give an overview of some of our APPL/A programs, including the use of language constructs in various roles to support various approaches to software-process modeling and implementation.

5.1 REBUS

REBUS is a program to support the process of specifying software requirements. REBUS was our first non-trivial process program, and it has been developed through several different versions. Early versions of the program were instrumental in evaluating early versions of APPL/A. In its current version, REBUS is running as part of the Arcadia demonstration, and it has been complemented by the development (also in APPL/A) of a comparable design process (DEBUS). The main results of this work are described in [67].

The goal of REBUS is to support the development of a requirements-specification product. The requirements product is represented in a graphical structure corresponding to a functional decomposition of requirements elements. Each element has required and optional fields. Nodes and edges of the requirements graph are stored in relations (consistent with the mathematical definition of graphs in terms of sets of nodes and edges). Relations prove quite flexible for this purpose since tuples representing optional fields need to be stored only for the elements that need them; the number of edges associated with a node can be made dynamically variable in a similar way. Predicates have been used to define desired properties of individual elements (e.g., each node ID is unique, all required fields are specified) and of the overall graph (e.g., no cycles). Triggers have been used for various purposes, including the propagation of updates and deletions between relations (e.g., deletion of a node triggers deletion of its stored attribute values) and the recording of execution histories. The programmability of APPL/A relation implementations has been especially useful in the different versions of REBUS. These have made use of Ada direct I/O files, the Cactis semantic database [30], and finally of the Triton object-management system (which was developed, in part, for this purpose) [28]. REBUS does not direct developers in the specification of requirements but it exploits APPL/A consistency-maintenance features to guide, support, and help manage users. One example of this is a locking mechanism so that interference between developers can be avoided.

REBUS also provides one of the initial illustrations of the need for a flexible consistency model. Our intention was that a complete and consistent requirements graph would be connected and acyclic. However, it did not seem reasonable that these conditions should necessarily hold (or that we should have to check for and enforce them) during development. For example, an initial graph might be developed in several pieces, perhaps by different developers. When the pieces were assembled, before the whole graph could be analyzed, cycles might occur. However, both of these conditions could be tolerated for an extended period during development. Thus it became apparent that we wanted constraints that could be enforced at different times in a process. It also became apparent that a constraint might be "turned on" in a state in which it was not satisfied, although that would not necessarily represent an error (e.g., if subsequent actions were to repair the inconsistency). This contributed to the motivation to tolerate inconsistency as a natural as well as an exceptional condition.

5.2 The ISPW6 Process

An APPL/A program was written for the process problem introduced for the Sixth International Software Process Workshop (ISPW6). This problem has been published in [42] and it has been addressed by numerous process researchers (e.g., [68, 41, 58]). Briefly, the process involves a unit change and test, and it comprises several managerial and engineering activities. The process is invoked in response to the modification of a requirement unit. It begins with a manager scheduling and assigning the various engineering tasks of the process. These include modification of the corresponding design unit, review of the modified design, modification of the corresponding code unit, modification of the test plan, modification of the unit test, and testing of the unit. During the execution of these activities the manager monitors their progress. The process description specifies various constraints on the process, such as a partial order on the execution of engineering tasks, the kinds of personnel responsible for each task, and conditions for task initiation and termination.

The APPL/A program for the ISPW6 process represents a partial implementation of the process; the program covers the whole process, but to varying levels of detail. The program has as its goal to demonstrate that APPL/A can be used to model the process, to implement the automatable parts of the process, and to automatically support the manual parts of the process. In doing this, the program addresses both process and product representation. All of the major constructs of the language are used in important roles.

APPL/A relations are used to store product data, e.g., latest and approved versions, and to represent relationships between different kinds of artifacts, e.g., the dependence of design units on requirements units. Relations are also used to store project and process data such as the members of the project team, the assignment of personnel to tasks, task start and completion dependencies, task start and finish schedules, task status (active and complete), and so on. APPL/A predicates specify a consistent state for the relations and could be enforced to assure that consistency holds. Predicates specify, for example, that each design unit depends on a valid requirement unit, that only project team members are assigned to tasks, that task start and finish times are consistent, and that the task schedule is consistent with specified task dependencies.

Ada packages are used to group related relations, predicates, and control constructs. For example, package Software_Products includes product-related declarations, and package Change_Process_Data groups declarations related to the definition and management of roles and activities within the change process. Packages Change_Management_Tasks and Change_Engineering_Tasks define Ada tasks that represent the management and engineering activities, respectively, in the change-and-test process. Ada procedures are used to represent the overall process in which these tasks are invoked and to represent steps within these tasks.

APPL/A triggers are not used directly to model the process but are used internally within the program to automate task invocation and termination and assist with process monitoring. The triggers monitor relations that represent task status and task schedules. In response to updates in these relations the triggers perform such actions as invoking new tasks when a prerequisite task has finished and notifying the manager when the deadline for completion of a task has arrived.

APPL/A consistency management statements, specifically the **serial**, **atomic**, and **suspend** statements, are used within the program to assure the consistency and integrity of composite updates to one or more relations. For example, the manager must schedule the engineering tasks and assign the required personnel to them; this must be done at the beginning of the process and possibly redone during the process. It requires that several mutually constrained relations be updated together. The use of the statements allows consistency to be relaxed while the updates are being made but assures that their overall effect will be consistent. The statements also prevent outside access to the relations while the updates are being performed; for example, triggers on the scheduled relations will not be activated until the final composite changes to the schedule are committed.

The APPL/A program for ISPW6 exhibits several interesting properties; among these are the following.

- Reactive control: Although elements of the process are imperative, major activities in the process (the managerial and engineering tasks, some sub-steps of the managerial task, and the process itself) are initiated (or terminated) in response to signals, events (like the completion of a task or modification of an artifact), temporal conditions, and exceptions.
- Process-state reification: Aspects of the state of the executing process, e.g., whether tasks are complete or executing, and when tasks are scheduled to be executed, are reified in the form of data stored in relations. This means that the program actually provides two representations of the process (one in code, i.e., commands, and one in data). The reification of process state supports reflexive behavior in the process, i.e., the program can examine the state of the process and act accordingly.
- Incorporation of meta-activities: The managerial tasks of the process represent meta-activities in that

the role of the managerial tasks is to control and monitor other tasks in the process. The APPL/A program for the ISPW6 problem thus represents not only the basic process of changing and testing a unit but also the meta-process of controlling and monitoring the basic process.

These same properties have shown up in many of our other process programs, and the ability to support them seems essential for a process-programming language.

5.3 ISPW7 Extensions to the ISPW6 Process

At the Seventh International Workshop on the Software Process (ISPW7) four extensions and variations on the ISPW6 process were introduced [27]. The APPL/A solutions for these are reported in [63]. Here we focus on the teamwork problem of coordination and communication among a team of developers, specifically, multiple programmers coding multiple inter-dependent modules.

The APPL/A program for this problem [63] represents modification of multiple code modules in parallel. It allows the number of modules (and, hence, of parallel activities) and also the specific inter-module dependencies to be determined and modified at run-time. Module-coding activities are represented by Ada tasks that are allocated dynamically as needed. Allocation is done by a trigger in response to changes in module status (e.g., a module being flagged incomplete or outdated). Relations are used to represent inter-module coding and compilation dependencies; enforced predicates (and corresponding organization of the control structure) assure that dependencies are observed by the process. The program allows merging of effort in the case of related modules, where any module may need to be adapted to accommodate any other module. Communication may be regularly scheduled or irregular and dynamically-determined. Coordination is supported by the sending of messages between process participants and by the sharing of relations that store products, data about the products and process, and comments provided by the participants. The consistency management statements are used to provide serializable, consistent, and atomic access to these relations by the concurrent tasks.

5.4 Management Process

MANLOBBI is an executable prototype process-program that has been developed to provide ISPW6-like management capabilities for the Arcadia demonstration environment. The program allows a manager to schedule and assign tasks, to set the level of enforcement of scheduling and assignment constraints, and to review and update the status of development activities. An important aspect of the design of this program is support for the reification of process and program state and the use of this reified state for a variety of purposes.

The program is designed so that the main activities in the management process are represented directly by the main control units of the management-process program. In other words, the high-level control architecture of the program reflects the control architecture of the process. This need not be true in all process programs [65], but it allows program state to serve as a surrogate for process state. The correspondence is maintained for three levels of abstraction: the main process and two levels of subtasks. (Below that, the activities tend to be too low-level or too generic to be very interesting from a process standpoint, e.g., I/O routines. Correspondingly, a non-hierarchical architecture becomes more appropriate for the program, even though it does not reflect process structure, because it allows the sharing of lower-level units and shortens and simplifies the program.)

The state of the program (representing also the state of the process) is reified in constrained relations. These relations record whether given parts of the process are active or complete[9] The procedures in the program that correspond to process steps are coded so that they update their status as they execute. When a procedure is entered, it sets its corresponding status to "active" and "incomplete." When a procedure is exited, it sets its corresponding status to "inactive" and to "complete" or "incomplete" according to whether the activity represented was finished. Predicates on the status relations specify, for example, that no first-level subtask can be complete unless all of its subtasks are complete, or that a later task cannot be active or complete before an earlier task is complete. The control flow in the program is designed to assure that these conditions are not violated under normal program execution, thus assuring that the process satisfies the same conditions.

In the APPL/A program for ISPW6, the reified process state was used for purposes of reflexive control. In the management process program, it serves this purpose and several others. Control is based on process state not only within a single program execution but also between program executions. The program is designed to be executed intermittently during the execution of the process, i.e., invoked and terminated multiple times at arbitrary points within the process. Whenever the program is invoked, it begins by assessing the state of the process as maintained persistently in the task-status relations. It then resumes executing at a point corresponding to the point in the process at which it left off. The ability to execute a process program intermittently like this is one approach to the support of long-term software processes.

MANLOBBI is the focus for a set of prototype process-engineering tools that are also programmed in APPL/A. MANVISUALIZER is a process visualization tool that monitors the status of tasks within MANLOBBI and displays that status for the manager. The manager can see which tasks are active, which have been completed, and which remain to be done. MANLOGGER is a logging tool that monitors and logs updates to the status of tasks within MANLOBBI. This log provides an execution trace of the process that can be used for such as purposes process discovery, measurement, and validation. The RASTA_MON is

[9]We recognize that there are other kinds of status that may be associated with a software process and process program. However, these are sufficient for purposes of this program, and they illustrate the relevant language features and their use in supporting this sort of process-program architecture.

a run-time anomaly spotting, tracking, and analysis monitor. This tool monitors updates to the process log and signals unusual or anomalous conditions in the execution sequence. These conditions include, for example, the restart of a task that has been declared complete, or the marking of a task complete when it has never been executed. SPLAT, a simple process-log analysis tool, provides some summary statistics for the process based on its execution log, including the number of times that each task and subtask in the process is executed, completed, and restarted. These tools depend heavily on shared relations and inter-process triggers. They demonstrate the utility of language features for supporting process-engineering applications.

5.5 Other Programs

APPL/A has been used to represent a variety of other software processes, process fragments, and tools. In the Arcadia demonstration system, APPL/A has been used not only in REBUS and DEBUS but also in several tools, including the PROCESS VIEWER, PROJECT CONTROL PANEL, and ARTIFACT CONNECTION VIEWER. An example of changes related to a code-and-compile process is presented in [66]. This addresses product change-management within a program. It also illustrates the modification of the program to introduce simple and more complex changes in the process, and it shows the use of another process program to facilitate the transition between different versions of the code-and-compile program. An early version of APPL/A was also used to code DataFlow⋈Relay, which integrates dataflow and fault-based testing and analysis [55].

6 Analysis and Evaluation

This section addresses various aspects of APPL/A for the purpose of understanding how, and how well, it works to achieve its design goals and to facilitate process programming. The issues identified and lessons learned relate both to APPL/A in particular and to process-programming languages in general.

6.1 Coverage of Design Goals

As discussed in Section 2, APPL/A must be judged, in part, by the extent to which it addresses its design goals. APPL/A succeeds in defining significant support for all of the capabilities required by these goals. However, there is not a one-to-one correspondence between language features and design goals. Here we identify the specific features of the language that address each of the goals:

- **Executability**: Derived from Ada (and the translation of APPL/A, which is largely into Ada).

- **Basic control and data definition capabilities**: Provided by Ada; APPL/A extensions to Ada are mostly related to other goals as described below.

- **Abstraction, encapsulation, and modularity**: Ada supports these, e.g., with separate specifications and bodies for packages and tasks. APPL/A draws on the Ada model for this in providing relations and triggers that have specifications and bodies analogous to those of Ada packages and tasks.

- **Representation of relationships**: APPL/A defines relations and tuples especially for this purpose. Derived attributes of relations are intended specifically for the representation of derivation relationships.

- **Derived data**: This is supported most specifically by the derived attributes of relations. Triggers can also be used, as a special case, to automatically derive data in response to updates to relations.

- **Data persistence and sharing**: Relations are persistent and shared.

- **Programmability of implementations**: The Ada-style separation of specifications and bodies allows implementations to be programmed for several kinds of constructs; APPL/A takes particular advantage of this in the programmability of relation implementations.

- **Concurrent and reactive control**: Concurrency is provided through Ada tasks and APPL/A triggers and relations. Triggers also provide reactive control.

- **Explicit, flexible, and dynamic consistency management**: APPL/A predicates provide for the explicit representation of enforcible conditions on relations. Flexibility and dynamism in consistency management are supported through the ability to turn the default enforcement of predicates on and off and by the ability of consistency-management statements to locally and temporarily control the actual enforcement of predicates.

- **Flexible transaction management**: This is provided by the consistency management statements, which individually offer a range of features related to serializability, atomicity, and predicate enforcement, and which can be nested and otherwise composed with other APPL/A and Ada control constructs. In combination with mechanisms associated with predicates, the consistency management statements provide considerable support for the accommodation of inconsistency.

APPL/A must also be judged by the degree to which the capabilities provided support effective process programs. As shown by the examples in Section 5, we have found that the language generally facilitates the implementation of process programs that effectively support the characteristics of software processes. More detailed evaluations of particular features of APPL/A in this regard are presented below.

6.2 Data Model

The use of relations has proven one of the more controversial aspects of APPL/A. It was not a goal of APPL/A research to develop a comprehensive data model for software products or processes. Rather, we believed that an enhanced relational data model would be particularly useful for representing software products and their interrelationships, and we sought to test this hypothesis. Relations also gave us the necessary basis for experimentation with other aspects of the language that relate to data, such as persistence, transactions, and triggers.

In our experience relations have indeed proven useful for representing software products and their interrelationships. As discussed in Section 5, they have also proven useful for representing aspects of software processes and projects. The usefulness of relations for our purposes derives from several properties of the data model. One is that relations and tuples are semantically neutral, i.e., no particular data-modeling semantics are ascribed to them. This affords flexibility in implementations because relations and tuples can be used in any of a variety of modeling roles, and they can be manipulated in a uniform manner regardless of their role. Another factor is that relations provide great flexibility in the representation of application objects. Query capabilities enable different views of objects to be obtained on an application-specific basis, and schemas are readily extensible because new attributes and relationships can be added for an object by adding new relations, i.e., without disturbing existing relations (or the applications that use them). Effective schema management and meta-data support are needed, however, to assure that systems composed of large numbers of relations and alternative views are kept consistent and understandable.

Our relational data model resembles the conventional relational data model [11] in that it is value-based. More specifically, data are passed into (and out from) relations by value rather than by reference. The alternative would have been a more object-based approach, where we take "object" to mean an entity (particularly, a data entity) that is accessed by reference. A value-based approach was taken for two reasons (despite our earlier work using an object-based data model [29]). One reason was that we wanted to conform to the Ada type model, so we did not introduce any stronger notions of object than are already present in Ada. In retrospect, this reason does not seem very important. The other reason was to avoid problems associated with the accessibility of objects via references. Object accessibility has serious complications for data modeling (discussed here) and also for triggering and consistency management (Sections 6.5 and 6.6, respectively).

Object accessibility via references is a problem for data modeling because it admits violations of abstraction. Abstraction is a generally important data-modeling principle and is a goal for APPL/A in particular with respect to relations. To see the problem, consider a reference-based object model in which instances of abstract data types (ADTs) are objects that are accessed via reference and that may contain other objects. If an object can be accessed via any reference to it, then the object can be modified regardless of whether it is contained in another object. For example, suppose object A is a stack, object B is a stack element, and B has been pushed onto A. Any holder of a reference to B can modify the state of B without performing any operation on A. However, such an operation on B also indirectly modifies the state of A, even though no operation is performed on A. Thus, the availability of references into the state of an ADT instance breaks the encapsulation of that state and effectively allows the access discipline of the ADT to be violated.

The problems that object accessibility causes for data abstraction in and of themselves were a significant concern for the choice of a data model for APPL/A. The further problems caused for triggering and predicate enforcement ultimately motivated our choice of a value-based data model. Our data model does not rely on the use of references that can lead directly into the state of objects of abstract types (specifically, relations). Rather, it provides for the manipulation of relation state only through operations in the relation interface, thus preserving the encapsulation of state and maintaining the desired access discipline.

6.3 Data Persistence

Support for persistent data has been essential for the representation of both software products (which was anticipated) and software processes (which was not). The particular persistence model used in APPL/A is based on the idea that all relations should be persistent so long as they are accessible. This model is admittedly simplistic, e.g., it does not allow for transient relations or for persistence to be determined dynamically. However, the model was chosen because it is simple and it was expected to be appropriate for most of our applications. In practice the model has served fairly well, but we have encountered a few cases in which transient relations would be useful. (For example, in the APPL/A solution for the ISPW7 resource-management problem [63], a local relation is used to store the values that represent keys for turning enforcement on or off. These keys are only needed within a single program execution. However, for reasons having nothing to do with persistence, relations still are an appropriate data structure for storing these values. That is because each capability must be related to a predicate, and it is helpful to be able to query the relation to obtain the key for a predicate.) Future enhancements of the persistence model should at least include provisions for transient relations, i.e., relations with lifetimes bound to those of the scopes in which they occur (and for completeness probably also "static" relations, i.e., relations with lifetimes bound to those of the program executions in which they occur). We have not had a need for a more elaborate persistence model.

6.4 Data Sharing

APPL/A requires that multiple programs have the ability to share data stored in a relation. Any program that includes the interface to a relation must be able to access the data in the relation, as if the program were one of (possibly) many operating on a common database. The relation implementation is required to support this sharing. However, relation interfaces are program library units, and the ability of a program to include them is governed by the visibility rules of the library system. In this way the space of shared relations can be structured, albeit by means outside of the scope of the language. The APPL/A model of data sharing thus has aspects related to both database and programming language systems.

The ability to share data between programs has proven essential, not just in support of concurrent engineering activities, but also for managerial oversight, process visualization and logging, process modification and repair, and intermittent (long-term) execution. Our model of data sharing is simple but general, and it has proven adequate for all of these purposes. Note that coordination of access to shared data is addressed by means of transaction-related capabilities (the consistency management statements in APPL/A). We have not addressed access control.

6.5 Triggers

APPL/A triggers are intended to address our language-design goal for reactive control, and they have proven very useful in this regard. Although triggers respond only to operations on relations, they otherwise serve as general-purpose control constructs. We have used them, for example, for monitoring product and process state, allocating tasks and sending notifications, and in support of process visualization. Their usefulness derives from the fact that they are reactive and concurrent. They enable actions to be taken when events or conditions dictate (e.g., logging product changes and modifying process displays as the data they represent are updated). They also facilitate the modular design of process programs by allowing actions that are logically separate from those of the main thread of control to be represented in a separate thread of control.

For syntactic and semantic uniformity, the APPL/A **upon** statement was modeled after the Ada **accept** statement. This entails synchronization between the relation and the trigger when a monitored event occurs. A consequence of this is that a trigger cannot *directly* call back to a relation from which it receives signals without incurring deadlock.[10] However, our experience suggests that it is sometimes useful for a trigger to call back to a relation to which it is responding (e.g., as when one update to a relation entails further updates for the maintenance of consistency). Thus the trigger model should be revised to allow triggers to respond asynchronously to relation events.

Triggers in APPL/A are event-based, that is, they respond to events associated with operations on relations. However, in the value-based data model used in APPL/A, all changes to the state of a relation must be made through operations defined in the relation interface. APPL/A triggers can therefore effec-

[10]Deadlock occurs because the trigger is suspended while its call is being accepted by the relation, and the relation is suspended while the signal for the acceptance of the call is propagated to the trigger.

tively track changes to relation state, and thus they can be used to achieve state-based triggering with respect to relations.

6.6 Predicates

As discussed in Section 2, and as shown in Section 5, consistency issues are often a primary consideration in our process programs. They affect the management of both software products and processes. Consequently, the incorporation of enforceable predicates into APPL/A has proven very useful. The ability to state consistency conditions explicitly facilitates program understanding, verification, maintenance, and reuse. This ability is also useful in process-program development. Our typical practice has been to design the relations and their constraints first, then to consider how the imperative constructs of the language may be used to achieve the goals of the process while satisfying the constraints. This separates the definition of consistent states within a process from issues of how the desired transitions between those states are to be achieved. Thus, the ability to express constraints has an important effect on process-program design style, which in particular leads to greater assurance that desired conditions will be satisfied.

The enforcibility of predicates is facilitated by the use of a value-based data model. Predicates must explicitly name the relations to which they apply; thus the predicates that apply to a relation are obvious, both to the programmer (or to analysis tools) and to the enforcement mechanism in the runtime system. Additionally, the data in a relation are accessible only through the abstract interface of the relation. Thus it is also obvious when a predicate should be checked, since it may only be violated by operations on the relations that it names.[11]

6.7 Transaction Model

Because our applications make use of persistent, shared data, some support for transactions is essential. The APPL/A transaction model has proven generally useful. Most use has been made of the serial, suspend, and atomic statements, often individually. Typical examples are found in the APPL/A program for ISPW6. The procedure to notify personnel affected by task schedules and assignments only needs to serializably read the relations in which the relevant data are stored. Atomicity is irrelevant, and there is no need for special consistency control. The Ada task that represents the managerial activity "monitor progress" includes atomic statements for composite updates to task status. Atomicity assures that either all or none of the updates are seen by the rest of the program (which is important in the

[11]With the current implementation, this is strictly true only for predicates that only reference relations. In general, a predicate cannot be violated by an operation on any relation to which it does not refer because updates to relations can be tracked and checked. However, updates to other data are not now monitored by the run-time system. Thus, if a predicate refers to some other data, then changes to those data may cause the predicate to be violated without alerting the enforcement mechanism.

control of dependent triggered activities), but again there is no need for special consistency control during the updates. The suspend statement by itself is also useful. For example, designs and related information about their versions are mutually constrained; when a new design is installed, the version information must also be updated also to maintain consistency. The suspend statement allows both the designs and version data to be updated together while the constraints on them are temporarily suspended. Examples such as these support our hypothesis that transaction-related capabilities such as concurrency control, atomicity, and control over predicate enforcement, are often useful independently or in various simple combinations. The enforce and allow statements have received less use. They are designed, though, for somewhat more specialized circumstances than the other statements (for example, see [66]). These circumstances have not been common in the particular process programs we have undertaken, but we continue to believe that such cases are important and should be supported.

The APPL/A program for the ISPW7 "teamwork" problem (described in Section 5.3 relies on APPL/A constructs to implement complex transactions that are customized to fairly elaborate, process-specific requirements. In separate, smaller exercises intended to test the flexibility and generality of the consistency-management statements, we have been able to use the statements to represent nested transactions (in which the commit of a child transaction is dependent on commit of the parent transaction, but not vice versa) [47], hierarchical transactions (in which the commit of a parent transaction is dependent on the commit of its children) [5], split transactions (in which the results of a single initial transaction are split and committed or aborted independently) [53], join transactions (in which the results of two separate initial transactions are committed or aborted together) [53], repositories with check-in and check-out, and other forms of transaction. We believe that such examples validate the viability of our approach to the programming of high-level transactions using compositions of simpler constructs. Some additional discussion of the APPL/A transaction model with respect to heterogeneous transactions (i.e., coordination of APPL/A transactions with other transactions) is presented in [70].

6.8 Accommodation of Inconsistency

The ability to accommodate inconsistency is supported by capabilities associated with both predicates and consistency management statements. The availability of these dual mechanisms is especially helpful since they have complementary aspects. The statements afford a consistency-control mechanism that is syntactically structured, definite in extent, and local to a program, while the predicate attributes offer a mechanism that is syntactically unstructured, indefinite in extent, and global (applying to all programs). The attribute-related mechanisms are thus well suited to long-term control of consistency enforcement and to adjustments of the default enforcement in response to exceptional inconsistencies. The statements provide a means for individual programs to adapt the enforcement regime, allowing local and temporary inconsistencies when data are otherwise consistent, or enabling work to be accomplished when data are otherwise inconsistent.

In our experience, the accommodation of inconsistency has been most important in the context of process change, when process or product data must be adopted and possibly adapted between different versions of a process (or different processes). Other sources of inconsistency have not usually been a problem for us, for a variety of reasons. First, since we are able to explicitly define consistent states for data (using predicates), we are generally able to design our programs so that they do not violate those predicates in normal execution. Consequently, inconsistency is seldom inherent in our individual processes. Second, we usually have substantial control over the execution environment of our programs; this has allowed us to exclude "outside" access to our data stores or other resources that might interfere with our programs and create inconsistencies. Finally, we have not developed many processes that have conflicting consistency requirements. Thus, where our processes share data, they usually also share common conditions for consistency of those data. The conditions that allow us to avoid inconsistency may occur in many other contexts, but we do not believe that they can be assured in general. Thus, we expect that inconsistency from various sources will be more typical than our experience suggests.

6.9 Interactions of Language Features

As noted in Section 2.3, part of the challenge of defining APPL/A was expected to be the integration of a wide range of different kinds of capabilities. This indeed was the case. A comprehensive discussion of the interactions of features in the language is beyond the scope of this paper. However, we give a few examples here to indicate the scope and significance of integration issues.

One problem that has already been mentioned is that of enforcing predicates on objects that contain other objects that can be referenced directly (see Sections 6.6 and 6.2). This issue also affects triggering (Section 6.5).

Another problem was the combination of triggers and transactions, specifically, what to do about the propagation of event signals from events occurring within transactions. If events are propagated immediately, triggers outside of the transaction may take actions based on results that are subsequently undone. Rather than allow this, or entirely preclude triggering off of events within transactions, we decided to hold up the propagation of signals from events within a transaction until the transaction had committed. This means that some triggers cannot synchronously respond to some events (a limitation on their desired functionality), but it does allow at least delayed triggering on events within transactions while avoiding triggering on "phantom" events (i.e., events that are subsequently undone).

Yet another interaction arose from the desire to make access to persistent data orthogonal to the use of transactions. Transactions need not be used for operations on APPL/A relations; however, issues such as consistency, recovery, and serializability remain relevant to relations. These concerns are usually supported through a transaction model. In order to free the use of relations from transactions, it was necessary to also associate these properties directly with relation operations.

6.10 Focus On Relations

Relations were made the focus of other new features in APPL/A because we believed that relations were an especially important kind of data structure for software processes and products and that focusing on relations would allow us to explore other issues (such as transactions and consistency management) in detail. As a research strategy these enabled us to uncover and address a number of issues in language design (e.g., the interactions of triggers and transactions), and, as the experience described in Section 5 indicates, the resulting language has proven usable and useful. The language is clearly unsatisfying, however, from the standpoint of type completeness. While type completeness was not one of our original goals, there would seem to be no *a priori* reason that the special features in the language (such as persistence and constraints) should not be extended to types other than relations. Our experience suggests that many such extensions are conceptually feasible, as has been demonstrated by the Pleiades system [70]. However, we would also be surprised if all such extensions were without consequences (e.g., as with the previously discussed issues of triggering and consistency management over objects).

6.11 Language-Based Process Support

An implicit premise of APPL/A is that software processes should be supported by capabilities provided to process programs through the process-programming language. However, many of the capabilities provided by APPL/A are also offered in some form as language-independent services (e.g., persistent storage, and "event management" [54, 72]). This raises the possibility that software processes might instead be supported in environments where conventional programming languages are supplemented with such external services. This can be an attractive prospect where an existing language must be used without modification [74]. Moreover, the language-based approach, in which needed capabilities are embedded in the language, has as a disadvantage that those capabilities may not be readily accessible to programs written in other languages. The advantage, though, is that the capabilities may be more closely integrated with other features of the language. The need to more closely integrate data types and transaction control into application-development languages was the main motivation for the development of persistent programming languages [2]. This rationale also motivated the development of APPL/A as a persistent programming language. APPL/A includes many other examples of interactions between features where there is a corresponding need for integration, such as between transactions and triggers and between triggers and relations. Because of the number of features provided by APPL/A, and the complexity of their interactions, the ability to integrate semantics and functionality has been critical to substantially satisfying a broad range of language-design goals.

7 Related Work

In this section we discuss work that is related to APPL/A, focusing primarily on languages and environments that are intended for the representation of software processes. We first give a broad categorization of process representation languages in order to place APPL/A in context within this diverse field. We then make more detailed comparisons to the languages to which APPL/A is most closely related.

7.1 Overview of Process Representation Languages

Process representation languages can be categorized roughly by the kind of process support they provide and the ways in which they provide it. Fernström [20] has suggested four categories of this sort: loosely coupled enactment, active process support, process enforcement, and process automation. Although the last two of these can be distinguished, our experience suggests that they are often combined in practice. We further recognize two additional categories: process modeling or simulation, and process implementation.

APPL/A, of course, is intended for process implementation. It is a more-or-less stand-alone programming language in which many aspects of executable process implementations may be programmed. Depending on process and process-program requirements, APPL/A programs may support any of the purposes indicated above. We have emphasized process support, enforcement, and automation, while at the same time attempting to address modeling.

At the other end of the spectrum from implementation languages are modeling and simulation languages that represent software processes primarily for purposes such as description and analysis. They typically do not, in and of themselves, support process execution. Examples of such languages include STATEMATE [24], MVP-L (multi-view process modeling language) [57, 56], LOTOS (language of temporal ordering specification) [58], and the Articulator [46]. Related to modeling and simulation is loosely coupled enactment. In this approach, a process model is used to guide participants through the process, but the process is executed or implemented separately from the system that supports the model. Coordination between actual development and the process model must be maintained manually, whereas coordination can be maintained automatically for languages that provide process support or implementation. None of the process languages reviewed here are intended for this purpose, although various stand-alone management tools (e.g., PERT charts) could be used in this way.

Systems that afford active process support provide access to development tools and data; they may also facilitate information flow, monitor process activities, and provide guidance to process participants.

Process WEAVER is one system that provides active process support [20]. Another is the process description language PDL [33] (based on context-free grammars) and its environment Hakoniwa [32]. A stronger level of process support is provided by languages and environments that enforce and automate software processes. These systems provide for a high degree of process control while still relying on external tools and infrastructure. These process-support systems are able to prescribe or proscribe development activities and to automate the machine-executable parts of a process. Examples of such systems include MELMAC [17] and SLANG [6], both of which are based on extended Petri-net models. Oikos [1] and Interact/Intermediate [50] may also be placed in this category. APPL/A programs may be written to provide active process support or process enforcement and automation. We have emphasized the latter, but requirements of the process and process program should determine the approach used. APPL/A is intended to allow access to outside programs and tools but also to support more complete implementations than are most of the languages in these groups.

Several languages and systems offer various kinds of support for full-scale process implementation. Some members of this category, like APPL/A, represent programming languages that can be used to code detailed process implementations. Others provide languages in combination with environments that may include such supporting systems as object management, version control, or configuration management. Other implementation languages and systems include HFSP, a hierarchical, functional language for programming software processes [39, 69], Merlin [52], a rule-based process-centered software environment, EPOS [15], a process environment supporting the reflexive, object-oriented process-modeling language SPELL [14], MARVEL [37, 35, 36], which supports the rule-based MARVEL Strategy Language and related object-management capabilities, and Adele-2 [7, 8], based on a central database with long transactions and triggers. AP5 [12, 13] is a system developed to support executable software specifications that has also been applied with some success to software process implementation [23].

A few points can be made about the spectrum of process-representation languages generally. They include a variety of computational models. The languages used for implementation tend to be some form of computationally-complete programming language (as APPL/A is based on Ada). Other languages are based on computational models, such as state-transition nets or grammars, that are computationally less powerful. However, these are usually enhanced in various respects and allow escape to external programs.

Kaiser and others [38] have proposed a classification of process languages based on the "level" of process that they are best able to support. "Global" languages facilitate the specification of the overall control flow and synchronization of a process, while "local" languages facilitate the expression of constraints on the invocation of individual tools or of operations on particular objects. HFSP is given as an example of a language that supports both levels of specification. They also state that the same can arguably be said of APPL/A. We believe that APPL/A does indeed support both levels of specification, although how apparent each of these may be depends on the way in which programs are designed and coded. More generally, the examples and criteria presented in [38] suggest that modeling and net-based languages tend to fall into the "global" category, while languages based on rules, triggers, or constraints tend to fall into the "local" category.

The implementation languages, including APPL/A, are primarily textual, as are many of the other languages. Graphical formalisms have been used for several purposes other than implementation. The distinction between graphical and textual languages can be blurred, though, by the use of graphical editors and visualization tools for textual languages and by underlying textual representations for graphical languages (compare, for example, Merlin, Oikos, and Process WEAVER).

7.2 Comparison with Implementation Languages

APPL/A is probably most comparable to HFSP in terms of approach in that both are language-oriented. APPL/A is more loosely coupled with the underlying environment than are the languages associated with Adele-2, EPOS, MARVEL, and Merlin. As an Ada-based language it has a very different sort of execution model and environment than AP5, which is based on Common Lisp. Below we compare some of the specific aspects of APPL/A with those in other languages. Some additional comparisons are made in Section 6.

APPL/A has an imperative control paradigm, supplemented with proactive and reactive concurrency, transactions, and exceptions. HFSP is centered on a functional control paradigm but also with concurrency, and Adele-2 relies heavily on triggers. Most of the other languages in this group are primarily rule-based. Many of the implementation languages are effectively multi-paradigm to some extent. However, our experience suggests that the balance of different kinds of control in APPL/A programs may be both more even or more variable than it is in many other languages. For example, rule-based (and trigger-based) languages typically include an imperative or procedural element; however, the rules or triggers are usually used to drive procedural or imperative components. In contrast, the control in an APPL/A program may be fairly evenly distributed over different kinds of components (as it is among subprograms, tasks, and triggers in the APPL/A program for ISPW6), or control may be vested in one or two kinds of components (as procedures alone might be sufficient for a linearly structured activity).

Meta-operations are an important part of several process implementation languages and systems. HFSP includes operations to create, destroy, suspend, and resume processes and to send and receive messages between processes. APPL/A defines no explicit operations for these purposes. Instead, we have relied on operating-system support for operations on executing programs and on Ada for operations on tasks within programs. The latter include operations to create and destroy tasks, while task entry calls provide inter-task communication. As a matter of practice, we have relied on entry calls to implement suspend and resume operations and also destroy operations in those cases where a task must perform some actions before it terminates. In SLANG and EPOS, meta-behavior is supported through a type hierarchy that represents process operations including operations that may modify process definitions. Merlin, Marvel, and AP5 also allow the representation of control elements that may affect other control elements. Process-state reification offers an alternative approach to meta-operations for purposes of process control since the "ordinary" operations of a language may support the representation of process state and be used to control a program and process accordingly.

In contrast with several of the process implementation languages, APPL/A triggers do not play a specialized role with respect to control. In particular, APPL/A does not differentiate between triggers that are used for maintaining consistency and triggers that are used to automate other activities. Such a distinction is made, for example, in AP5 [12, 13] and MARVEL [37, 35, 36], which have consistency rules (to repair constraint violations) and automation rules (for general processing). Adele-2 [7] distinguishes different kinds of triggers that can be used like consistency and automation rules. APPL/A triggers, although of a single kind, may be used for either purpose, as may the other control constructs in the language. Adele-2 [8], like APPL/A, also distinguishes types of triggers called global and local. However, in Adele-2 the distinction relates to whether the affected data are visible in the environment of the trigger, whereas in APPL/A the distinction relates to whether the triggering operation is within the scope in which the trigger is defined.

The issue of consistency versus automation triggers relates to the question of how triggers interact with transactions, especially with respect to rollback. Consistency rules typically are performed as part of transactions with which they are associated, i.e., the rules are invoked within a transaction to repair consistency violations caused by the transaction, and the results of the rules are committed (or aborted) if and only if the transaction is committed (or aborted). Automation rules typically are based on the committed results of a transaction and entail the execution of a new transaction. A similar situation applies in Adele-2 to pre- and post-triggers (which are part of a transaction) and after- and abort-triggers (which represent new transactions). HiPAC [45] provides classes of rules that allow triggered actions to be performed as a part of a given transaction, as a subtransaction of a given transaction, or as a separate top-level transaction. In APPL/A these effects can be achieved syntactically (by nesting triggers and CM statements) and by the use of **detached** operations and CM statements.

The APPL/A transaction model is unique with respect to those of other process implementation languages (and the process languages generally). As noted in Section 2.2, conventional database transactions (flat, atomic, serializable) are widely regarded as inadequate for software engineering applications. The APPL/A transaction model overcomes this limitation by providing a group of relatively simple statements that can be combined in various ways to create alternative forms of high-level transaction according to process-specific requirements. Some process implementation languages and systems (e.g., EPOS and Adele-2) support particular forms of long-term transactions based on local workspaces with check-in/check-out. Marvel supports alternative concurrency-control policies which are implemented through a central concurrency controller rather than by application-specific programming. AP5 and Pleiades [70], which have comparable kinds of features to APPL/A but which are intended for other purposes, both have relatively conventional, flat transaction models.

The toleration of inconsistency has also been advocated by Balzer [4]. His approach, in the context of AP5, is to provide automated guards to flag data that violate consistency conditions. These guards can be tested by applications that depend on the satisfaction of the conditions. In APPL/A, no guards are set to flag predicates that are violated, although predicates themselves can be tested. On the other hand, in APPL/A the enforcement of predicates is automatic, and applications that require consistency are protected against inconsistency regardless of whether they explicitly test for it. Pleiades has adopted a model of optionally enforcible predicates that is based on that in APPL/A.

8 Summary and Conclusion

APPL/A is a prototype software-process programming language. It is intended to define control and data models appropriate for process programming, investigate the feasibility of integrating these models, confirm hypothesized requirements on process-programming languages, and provide a vehicle for experiments in process programming. APPL/A makes four main kinds of extensions to Ada: shared persistent relations, concurrent, reactive triggers, optionally enforcible predicates, and five transaction-like composite statements. The predicate and transaction models are particularly distinctive, and they provide the additional ability to accommodate inconsistency.

The constructs provided by APPL/A have enabled us to program a wide variety of kinds of software process. These include engineering-oriented development tasks as well as managerial processes. We have also used APPL/A to program a variety of other tools and parts of the APPL/A runtime system. APPL/A has further enabled us to address many different characteristics of software processes and to explore various aspects of process programming. These include such things as process, product, and project management, long-term and intermittent activities, cooperative work, the integration of manual and automated tasks, program design for process visualization, monitoring, and measurement, approaches to process change, the utility of process-state reification, and the importance of requirements, design, and other phases of the process-software life-cycle.

Our reliance on the new constructs in our process programs implies that programming languages for conventional applications, while they may provide important capabilities, are not yet really practical for software-process applications. Our ability to address a wide range of processes and process characteristics leads us to believe that we have identified an additional set of important, general, and powerful language capabilities for software-process programming.

Acknowledgements

This work has benefited from the comments of many people over a period of years. Most important are the members of the Arcadia consortium generally, who have offered many useful suggestions. Mark Maybee and Xiping Song in particular were helpful in providing feedback on experience with the language, especially in its earlier forms. Peri Tarr deserves thanks for detailed technical discussions on the issues addressed by APPL/A and for reading earlier drafts of this manuscript. We also wish to thank members of the software process community generally for their comments on APPL/A and the anonymous reviewers of previous APPL/A papers for their thoughtful suggestions on the presentation of this material.

References

[1] V. Ambriola, P. Ciancarini, and Montangero. Software process enactment in oikos. In *Proc. of the Fourth ACM SIGSOFT Symposium on Practical Software Development Environments*, pages 183–192, 1990. Irvine, California.

[2] Malcolm P. Atkinson, Peter J. Bailey, K. J. Chisholm, W. P. Cockshott, and Ronald Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.

[3] D.A. Baker, D.A. Fisher, and J.C. Shultis. *The Gardens of Iris.* Incremental Systems Corporation, Pittsburgh, PA, 1988.

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

[5] F. Bancilhon, W. Kim, and H. Korth. A model of CAD transactions. In *Proc. of the Eleventh International Conf. on Very Large Databases*, 1985.

[6] 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.

[7] Noureddine Belkhatir, Jacky Estublier, and Melo L. Walcelio. Adele 2: A support to large software development process. In *Proc. of the First International Conference on the Software Process*, pages 159 - 170, 1991. Redondo Beach, California, October, 1991.

[8] 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.

[9] R. F. Bruynooghe, J. M. Parker, and J. S. Rowles. PSS: A system for process enactment. In *Proc. of the First International Conference on the Software Process*, pages 128 - 141, 1991. Redondo Beach, California, October, 1991.

[10] Geoffrey M. Clemm and Leon J. Osterweil. A mechanism for environment integration. *ACM Transactions on Programming Languages and Systems*, 12(1):1–25, January 1990.

[11] E. F. Codd. A relational model for large shared data banks. *Comm. ACM*, 13(6):377–387, 1970.

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

[13] Donald Cohen. Compiling complex database transition triggers. In *Proceedings ACM SIGMOD '89 International Conf. on Management of Data*, pages 225 – 234, 1989.

[14] R. Conradi, M. L. Jaccheri, C. Mazzi, A. Aarsten, and M. N. Nguyen. Design, use, and implementation of SPELL, a language for software process modeling and evolution. In J.-C. Derniame, editor, *Proc. EWSPT '92, Trondheim, Norway.* Springer Verlag LNCS, September 1992.

[15] R. Conradi, E. Osjord, P. H. Westby, and C. Liu. Initial softare process management in EPOS. *Software Engineering Journal (special issue on software process and its support)*, pages 275 – 284, September 1991.

[16] Reidar Conradi, Chunnian Liu, and M. Letizia Jaccheri. Process modeling paradigms: An evaluation. In *Proc. 7th International Software Process Workshop*, 1991. Yountville, California.

[17] Wolfgang Deiters and Volker Gruhn. Managing software processes in the environment melmac. In *Proc. of the Fourth ACM SIGSOFT Symposium on Practical Software Development Environments*, pages 193–205, 1990. Irvine, California.

[18] Mark Dowson. Panel introduction – Why is process important? In *Proc. of the First International Conference on the Software Process*, page 2, 1991. Redondo Beach, California, October, 1991.

[19] Stuart I. Feldman. Make – a program for maintaining computer programs. *Software – Practice and Experience*, 9:255 – 265, 1979.

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

[21] Christer Fernström and Lennart Ohlsson. Integration needs in process enacted environments. In *Proc. of the First International Conference on the Software Process*, pages 142 – 158, 1991. Redondo Beach, California, October, 1991.

[22] Hector Garcia-Molina, Dieter Gawlick, Johannes Klein, Karl Kleissner, and Kenneth Salem. Modeling long-running activities as nested sagas. *IEEE Technical Committee on Data Engineering*, 14(1):14–18, March 1991.

[23] Neil Goldman and K. Narayanaswamy. Solution to the ISPW6 process examle. Distributed at the Sixth International Software Process Workshop, 1991.

[24] 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.

[25] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[26] Dennis Heimbigner. APT: APPL/A to ada translation. Arcadia Document CU-89-11, University of Colorado, Department of Computer Science, Boulder, Colorado 80309, 1989.

[27] Dennis Heimbigner. The process modeling example problem and its solutions. In *Proc. of the First International Conference on the Software Process*, page 174, 1991. Redondo Beach, California, October, 1991.

[28] 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.

[29] Dennis Heimbigner, Leon J. Osterweil, and Stanley M. Sutton, Jr. Active relations for specifying and implementing software object management. Technical Report CU-CS-406-88, University of Colorado, Department of Computer Science, Boulder, Colorado 80309, July 1988.

[30] Scot Hudson and Roger King. CACTIS: A database system for specifying functionally-defined databases. In *Proc. of the Workshop on Object-Oriented Databases*, pages 26–37, September 1986.

[31] Watts S. Humphrey. Panel position statement – why is process important? In *Proc. of the First International Conference on the Software Process*, page 3, 1991. Redondo Beach, California, October, 1991.

[32] 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.

[33] H. Iida, Takeshi Ogihara, K. Inoue, and K. Torii. Generating a menu-oriented navigation system from formal description of software development activity sequence. In *Proc. of the First International Conference on the Software Process*, pages 45 – 57, 1992.

[34] R. Kadia. Issues encountered in building a flexible software development environment – lessons from the Arcadia project. In *Proc. of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 169–180, 1992.

[35] Gail E. Kaiser. Rule-based modeling of the software development process. In *Proc. 4th International Software Process Workshop*, October 1988. Published in ACM SIGSOFT Software Engineering Notes, v. 14, n. 4, June, 1989.

[36] Gail E. Kaiser, Naser S. Barghouti, and Michael H. Sokolsky. Preliminary experience with process modeling in the marvel software development environment kernel. In Bruce D. Shriver, editor, *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 131–140, 1990. Kona, Hawaii, January, 1990.

[37] Gail E. Kaiser and Peter H. Feiler. An architecture for intelligent assistance in software development. In *Proc. Ninth International Conference on Software Engineering*, pages 180 – 188, 1987.

[38] Gail E. Kaiser, Steven S. Popovich, and Ben-Shaul Israel Z. A bi-level language for software process modeling. In *Proc. of the 15th International Conference on Software Engineering*, pages 132–143, 1993.

[39] Takuya Katayama. A hierarchical and functional software process description and its enaction. In *Proc. of the 11th International Conference on Software Engineering*, pages 343 – 353, 1989.

[40] Marc I. Kellner. Multiple-paradigm approaches for software process modeling. In *Proc. 7th International Software Process Workshop*, 1991. Yountville, California.

[41] Marc I. Kellner. Software process modeling support for management planning and control. In *Proc. of the First International Conference on the Software Process*, pages 8 – 28, 1991. Redondo Beach, California, October, 1991.

[42] Marc I. Kellner, Peter Feiler, Anthony Finkelstein, Takuya Katayama, Leon J. Osterweil, and Maria H. Penedo. ISPW-6 software process example. In *Proc. of the First International Conference on the Software Process*, pages 176 – 186, 1991. Redondo Beach, California, October, 1991.

[43] Manny M. Lehman. Panel position statement – why is process important? In *Proc. of the First International Conference on the Software Process*, page 4, 1991. Redondo Beach, California, October, 1991.

[44] Mark J. Maybee, Dennis H. Heimbigner, David L. Levine, and Leon J. Osterweil. Q: A Multi-lingual Interprocess Communications System for Software Envi ronment Implementation. Technical Report CU-CS-476-92, University of Colorado, Department of Computer Science, June 1992.

[45] Dennis R. McCarthy and Umeshwar Dayal. The architecture of an active data base management system. In *Proc. of the ACM SIGMOD International Conf. on the Management of Data*, pages 215 – 224, 1989.

[46] Peiwei Mi and Walt Scacchi. Modeling articulation work in software engineering processes. In *Proc. of the First International Conference on the Software Process*, pages 188 – 201, 1991. Redondo Beach, California, October, 1991.

[47] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Massachusetts Institute of Technology, May 1981.

[48] Leon J. Osterweil. Software processes are software, too. In *Proc. Ninth International Conference on Software Engineering*, 1987. Monterey, CA, March 30 – April 2, 1987.

[49] Leon J. Osterweil. Panel position statement – why is process important? In *Proc. of the First International Conference on the Software Process*, page 5, 1991. Redondo Beach, California, October, 1991.

[50] Dewayne E. Perry. Policy-directed coordination and cooperation. In *Proc. 7th International Software Process Workshop*, 1991. Yountville, California.

[51] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[52] Burkhard Peuschel, Wilhelm Schäfer, and Stefan Wolf. A knowledge-based software development environment supporting cooperative work. *International Journal on Software Engineering and Knowledge Engineering*, 1992. to appear.

[53] Calton Pu, Gail E. Kaiser, and Norman Hutchinson. Split-transactions for open-ended activities. In *Proc. of the Fourteenth International Conf. on Very Large Data Bases*, pages 26 – 37, 1988.

[54] Steven P. Reiss. Connecting tools using message passing in the field environment. *IEEE Software*, July 1990.

[55] Debra J. Richardson, Stephanie Leif Aha, and Leon J. Osterweil. Integrating testing techniques through process programming. In *Testing, Analysis, and Verification (3)*, pages 219–228, Key West, December 1989. SIGSOFT.

[56] H. D. Rombach. MVP-L: A language for process modeling in-the-large. Technical Report CS-TR-2709, Dept. of Computer Science, Univ. of Maryland, College Park, 1991.

[57] 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.

[58] Motoshi Saeki, Tsuyoshi Kaneko, and Masaki Sakamoto. A method for software process modeling and description using LOTOS. In *Proc. of the First International Conference on the Software Process*, pages 90 – 104, 1991. Redondo Beach, California, October, 1991.

[59] Richard W. Selby, Adam A. Porter, Doug C. Schmidt, and Jim Berney. Metric-driven analysis and feedback systems for enabling empirically guided software development. In *Proc. of the 13th International Conference on Software Engineering*, pages 288 – 298, May 1991. Austin Texas, May 13 – 17, 1991.

[60] Xiping Song and Leon J. Osterweil. Comparing design methodologies through process modeling. In *Proc. of the First International Conference on the Software Process*, pages 29 – 44, 1991. Redondo Beach, California, October, 1991.

[61] Kevin J. Sullivan and David Notkin. Reconciling environment integration and component independence. In *Proc. of the Fourth ACM SIGSOFT Symposium on Practical Software Development Environments*, pages 22–33, 1990. Irvine, California.

[62] Stanley M. Sutton, Jr. *APPL/A: A Prototype Language for Software-Process Programming.* PhD thesis, University of Colorado, August 1990.

[63] Stanley M. Sutton, Jr. APPL/A solutions for the ISPW7 process-modeling problems. Arcadia Document CU-91-07, University of Colorado, Department of Computer Science, Boulder, Colorado 80309, October 1991.

[64] Stanley M. Sutton, Jr. A flexible consistency model for persistent data in software-process programming languages. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Implementing Persistent Object Bases – Principles and Practice*, pages 305–318. Morgan Kaufman, 1991.

[65] Stanley M. Sutton, Jr. Opportunities, limitations, and tradeoffs in process programming. In *Proc. of the Second International Conference on the Software Process*, pages 135–146, 1993.

[66] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. Managing change in software development through process programming. Technical Report CU-CS-531-91, University of Colorado, Department of Computer Science, Boulder, Colorado 80309, June 1991.

[67] Stanley M. Sutton, Jr., Hadar Ziv, Dennis Heimbigner, Harry Yessayan, Mark Maybee, Leon J. Osterweil, and Xiping Song. Programming a software requirements-specification procss. In *Proc. of the First International Conference on the Software Process*, pages 68 – 89, 1991. Redondo Beach, California, October, 1991.

[68] Masato Suzuki, Atsushi Iwai, and Takuya Katayama. A formal model of re-execution in software process. In *Proc. of the Second International Conference on the Software Process*, pages 84–99, 1992.

[69] 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.

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

[71] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon J. Osterweil, Richard W. Selby, Jack C. Wileden, Alexander Wolf, and Michal Young. Foundations for the Arcadia environment architecture. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1 – 13. ACM, November 1988.

[72] Gary Thunquest. Supporting task management & process automation in the SoftBench development environment. In *Proc. 7th International Software Process Workshop*, 1991. Yountville, California.

[73] United States Department of Defense. *Reference Manual for the Ada Programming Language*, 1983. ANSI/MIL-STD-1815A-1983.

[74] Alex L. Wolf. An initial look at abstraction mechanisms and persistence. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Implementing Persistent Object Bases – Principles and Practice*, pages 360–368. Morgan Kaufman, 1991.

[75] Alex L. Wolf and David S. Rosenblum. A study in software process data capture and analysis. In *Proc. of the Second International Conference on the Software Process*, pages 115–124, 1983.