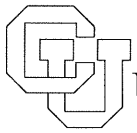
Managing Change in Software Development Through Process Programming

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

CU-CS-531-91 June 1991



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Managing Change in Software Development through Process Programming[1]

Stanley M. Sutton, Jr.[2] Dennis Heimbigner Leon J. Osterweil[3]

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

CU-CS-531-91
June, 1991

Abstract

Change is pervasive during software development. Change management can be facilitated by software-process programming, which formalizes software products and processes in software-process programs. Toward this end, process-programming languages (PPLs) should include constructs that address specific change-management problems. These include lack of explicit representation for relationships, weak or inflexible constraints on objects and relationships, visibility of implementations, lack of formal representation of processes, and dependence on manual practices.

APPL/A is a prototype PPL that addresses these problems. APPL/A is an extension to Ada. APPL/A includes abstract, persistent relations with programmable implementations, relation attributes that may be composite and derived, triggers that react to relation operations, optionally-enforcible predicates on relations, and five composite statements that provide flexible transaction-related capabilities.

Relations enable relationships to be represented explicitly and derivation dependencies to be maintained automatically. Relation bodies may implement alternative storage and computation strategies without affecting users of relation specifications. Triggers can automatically propagate data, invoke tools, and perform other change-management tasks. Predicates and the transaction-related statements can be used to support change management in the face of concurrent processes and evolving standards of consistency. Together, these features mitigate many of the problems that complicate change management in software development.

[1]Submitted to ACM Transactions on Software Engineering and Methodology.
[2]Author to whom correspondence should be addressed. Telephone: (303) 492-7906; email: sutton@cs.colorado.edu.
[3]Department of Information and Computer Science, University of California, Irvine, CA 92717

1 Introduction

Change is pervasive during software development. The components of a software product change: requirements, design, code, test cases, and so on are created, derived, debugged, and modified. The development process can change: A new phase may be added (e.g., prototyping), or an existing phase may be revised (e.g., providing more stringent testing). The supporting environment can also change: new tools may be added, and major components such as the underlying storage system may be replaced. Often change management is relegated to the maintenance phase of software development, but in practice "maintenance" (as a synonym for managing change) occurs throughout the software life cycle [16].

One major difficulty in managing change is in detecting and propagating the effects of a change to other components or aspects of the environment. A change to one object often requires changes to other objects that are derived from it or that must be kept consistent with it, and changes to those objects must be propagated in turn. A change in the development process must be implemented consistently and correctly in the face of existing practices, tools, and products. A change to parts of an environment can force adaptations in processes and other parts, with further potential for errors and inconsistencies in these areas.

In a process-programming environment, the process program serves as a focal point and integration mechanism for development activities and their supporting technologies. Process programs are encodings of software processes in formal process-programming languages [27]. Examples of environments which are intended to support process programming or which are driven by process programs include Arcadia [44], E-L [8, 9], ASPECT [22], Melmac [14], Oikos [2], and the software factory system described in [23]. Software-process programming has the potential to make the change-management problem tractable by formalizing the structure of software products and the processes by which they are constructed and maintained. By making this structure explicit, a process program can enable the tracking of changes. Moreover, it offers the possibility that the process of change propagation can be automated. With respect to change management, process programming is somewhat similar to software configuration management [15, 10, 18]. However, process programming languages are intended to represent a wider range of objects and processes than are conventional configuration-management systems.

If the potential of software-process programming is to be realized we must develop software-process programming languages with appropriate constructs and capabilities. Requirements for software-process programming languages (PPLs) are difficult to determine *a priori*. It seems reasonable to assume that PPLs must subsume the capabilities of conventional programming languages. However, we also expect that PPLs will include extensions and specializations that reflect the distinctive aspects of software processes and products. These should include elements relevant to change management. Given such a PPL, it should be possible to write process programs in which change is managed integrally and effectively.

In this paper we argue that process programming can indeed provide the kinds of support for change management suggested above. We first describe APPL/A [41], a prototype PPL based on Ada [45]. APPL/A is one part of the process-programming research taking place in the Arcadia project [44]. APPL/A provides basic constructs and capabilities that can be applied in various ways to support change management in process programs. We then present an extended example of an evolving process program that illustrates features of APPL/A and shows how change can be represented and managed in the context of process programs and process programming.

The remainder of paper is organized as follows. Section 2 presents a scenario that illustrates several common kinds of change in a software environment. On the basis of this scenario we draw some conclusions about what makes change management difficult and recommend language capabilities to make it easier. Section 3 provides an overview of the APPL/A programming language and discusses how features in the language provide support for change management. Section 4 provides an extended example. This example shows how change can be managed within an APPL/A process program and also shows how change to the process can be implemented through change to the program. The paper concludes with a discussion of related work and a summary of the status of APPL/A research.

2 A Scenario of Change

Many changes in software environments can be classified into three categories:

- Changes to objects in the environment
- Changes to processes supported by the environment
- Changes to the environment itself, including tools, support systems, and hardware

The following scenarios illustrate changes of these kinds and the resulting problems. The examples have been kept simple, but the problems they illustrate are nevertheless fundamental and widespread in software development.

The hypothetical system in which the scenarios are set is a simple development environment consisting of several tools in a UNIX-like operating system. The initial tools include a compiler, a loader, and a dataflow analyzer capable of detecting anomalies such as uninitialized variables, unused variables, etc. [26, 25].

The development process is informal. Programmers write source code, compile this to object code, link object code to executable code, then test the resulting executable code for bugs. If bugs are found the source code is revised and the process repeated. The dataflow analyzer may be used occasionally during the writing of source code (in an attempt to avoid errors) or when debugging (in an attempt to identify the source of errors).

This scenario is, of course, simplistic. Many projects would use tools such as Make [15] or SCCS [32] to help manage changes to code. Such tools help to manage certain kinds of change but not others; for example, they typically rely on a fixed storage system, they focus on derivation relationships, they use a fixed evaluation and caching strategy, and they provide limited inferencing capabilities. As we hope to show in this scenario and in the example of Section 4, effective change management demands on more general and flexible capabilities.

2.1 Changes to Objects

The normal course of development in this environment involves repeated additions, updates, and deletions of source modules. Suppose a programmer adds a new source module. She or he must then determine how to proceed. There are few constraints on the process. The programmer may first invoke either the dataflow analyzer or compiler; the use of either may or may not be conditioned on the results of the other. The dataflow analyzer may be ignored altogether, perhaps because the programmer does not care to use it or is simply unaware of it. Once a plan for tool invocation is determined it must be carried out manually. For example, the programmer may apply the dataflow analyzer to the source code, evaluate the results of the analysis, and then apply the compiler if the analysis is acceptable. Note that both the compiler and the dataflow analyzer create new objects whose types are distinct from the type of the source code and that the application of these tools creates an implicit dependency relationship between the source code and the new objects.

The programmer must also iterate this process for new objects derived from the source code. It is necessary to check whether a compilation is successful and, if so, link the resulting object module into the executable modules to which it belongs. The effects of these changes also need to be propagated further, for example, to the rederivation of test results. In a more realistic situation there might be still more tools that apply to any new derived object.

A similar situation occurs if an existing source module is updated. The applicable tools must be identified and invoked, and the resulting changes to derived objects must be propagated. In this case, however, the programmer is also responsible for identifying and removing previous versions of various kinds of objects (either deleting or archiving them) and for maintaining the consistency of system configurations that combine various versions of various modules.

The problems indicated above may arise for an individual programmer; these problems are compounded for teams of programmers. Access to and updates of objects may not be coordinated. For example, two programmers may separately edit the same source-code module at the same time, and one may overwrite the other's changes. Similarly, one programmer may attempt to recompile a module that is is being edited by another, or to relink an executable system where object modules are out-of-date because of changes made by others. Different modules or systems may be developed by different methods. For example, one programmer may rely on dataflow analysis, while another ignores it. And different programmers may handle out-dated objects in different ways. One may save out-dated versions, while another simply overwrites them. Thus the problems associated with individual programmers are multiplied, and new problems arise from the interactions of teams of programmers.

2.2 Changes to Processes

The processes by which software is developed are subject to change for many reasons. For example, suppose that the project manager institutes a policy requiring that all source modules must meet certain criteria with respect to data flow before they can be compiled. This implies that the dataflow analyzer and compiler should be applied to the source modules in sequence and that the application of the compiler is conditional on the results of the analysis. In order to implement this policy, programmers must be aware of the sequence and must understand the conditions under which compilation is allowed. They must also manually carry out the prescribed process.

The availability of tools to implement a given development process does not guarantee that it will be carried out consistently or correctly, however. The likelihood that a process will be executed improperly increases when the process is changed. In this scenario the change of process is small and the resulting process is simple. However, the potential remains for human error at several points. This potential increases as the complexity of the process and the magnitude of changes increase. Additionally, as with problems arising from changes to objects, the problems arising from changes to process are made worse when teams of developers are

involved.

2.3 Change to the Environment

Environments can change in many ways, each with consequent problems in change management. One common change to software environments is the addition of a new tool. For example, suppose the environment above is extended to include a "word-count" tool similar to the UNIX "wc" tool that counts the number of lines, words, and characters in given files. In this scenario the addition of the tool is simple because there are no restrictions on how and when it can or should be used.

At first this new tool may not be widely or effectively used. Programmers may not be aware of it, they may not understand its function and relationship to existing tools and objects (admittedly simple in this case), or they may not see any need for it. Eventually some programmers may begin to use it occasionally, possibly to measure their productivity or to obtain information on the size of modules as an aid in managing program complexity. Even so the tool may still not be used consistently or comprehensively. Finally, the project manager may promulgate a new policy that requires the size of all source code modules to be within certain limits. The word-count result for each source module is to be saved along with the source modules for subsequent review by management. This makes the role of the tool more specific, but it requires programmers to change their work habits. It also requires management of a new type of object, the word-count results, which must be stored and kept consistent with the source modules.

2.4 Causes and Consequences of Change-Management Problems

The scenarios presented here illustrate only some kinds of change. Many other kinds can occur: changes to hardware, resources, and personnel, among others. However, these scenarios are indicative of the kinds of problems that can result from change. These problems are attributable to several fundamental and interdependent causes:

- **Manual management of change**. Depending on programmers for change management entails the potential for human error. This may result in incomplete, incorrect, inconsistent, and inefficient response to changes.
- **Lack of explicit representation for relationships between objects**. Without some such representation, when one object is changed, it is difficult to identify which other objects are affected. The direction and extent of change propagation are difficult to determine, and changes may not be propagated completely. (Note that inter-object relationships include dependencies that are established both by automated derivations and manual activities.)
- **Lack of information about constraints on objects and their relationships**. Without some form of constraints, and in the absence of specifications for object derivations, it may be difficult to understand the consequences of any given change, and therefore difficult to propagate the effects of changes correctly and efficiently.
- **Dependence of the development process on implementation factors**. Although the development process in the abstract should be independent of implementation factors, changes in supporting systems can nevertheless force changes in the development process. These in turn may lead to problems in the process and resulting product.
- **Lack of explicit representation of the development process**. Developers may lack a clear, correct, and consistent understanding of the development process. Consequently the management of change within the process may be incomplete and inefficient. Moreover, a lack of representation makes the process itself difficult to change and increases the likelihood that changes will be carried out incorrectly and inconsistently. A particularly important aspect of this problem is lack of support for coordinating activities among members of a team of developers.

2.5 Recommendations for PPLs

We believe that an appropriately designed software-process programming language can alleviate many problems in change management. Certainly, the programming of processes in a PPL directly addresses the need for explicit representation of the development process. Additionally, the use of appropriate language constructs in formalizing the processes and products can help with the other problems associated with change. We recommend that PPLs should provide the following capabilities:

- Explicit representation of both objects and inter-object relationships.
- Explicit representation of the semantics of objects and relationships, including constraints and derivations.
- Support for coordination of concurrent activities, including concurrent access to persistent objects.

- Automation of as much of the change process as is feasible, including propagation of data, maintenance of consistency, and invocation of tools.
- Abstraction of processes, objects, and relationships from the underlying implementation system. At the logical level change management should be independent of the implementation, and changes to the implementation should not affect the abstract representation of development processes and products.

We believe that these items comprise a set of basic PPL requirements in the area of change management. In the next section we present an overview of APPL/A and show how its features address these requirements.

3 APPL/A

APPL/A is a prototype PPL [39, 42]. It is defined as an extension to Ada [45]. Ada provides the general-purpose capabilities that we believe any PPL must include. APPL/A has additional features that address the special needs of software process programming, including change management, data modeling, derived data, persistent data, consistency management, and accommodation of inconsistency.

The principal extensions that APPL/A makes to Ada include programmable persistent relations, triggers on relation operations, optionally-enforcible predicates on relations, and several composite statements that support the synthesis of a wide range of transaction-related constructs. The use of APPL/A constructs to support change management is discussed further below and illustrated in the example in Section 4.

3.1 Relations

Relation units in APPL/A provide for the storage of persistent data. APPL/A relations, like relations in conventional relational databases, represent the abstract mathematical notion of a relation, i.e. a subset of the cross-product of a list of object domains. However APPL/A relations have several important differences from the relations of conventional databases. APPL/A relations can have composite and abstract attribute types, they can have derived attributes, and they have programmable implementations. These extensions of the conventional relational model [11] make it more appropriate for software-object management and for change management in particular. Some other recent projects which implement relations include Postgres [33, 37], an advanced data-management system, and AP5, which extends Common Lisp with relations [12].

```
Relation Source_Repository is
-- Stores source modules with related data
--
    type src_repo_tuple is tuple
        author: in name_type;
        name: in name_type;
        src: in source_code;
    end tuple;
entries
    insert(author: in name_type;
        name: in name_type;
        src: in source_code);
    delete(author: in name_type;
        name: in name_type;
        src: in source_code);
    update(author: in 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);
    find(iterator: in out integer;
        first: boolean;
        found: out boolean;
        t: out src_repo_tuple;
        select_author: boolean;
            author: name_type;
        select_name: boolean;
            name: name_type;
        select_src: boolean;
            src: source_code);
End Source_Repository;
```

Figure 1: Sketch of Specification for Relation Source_Repository

Syntactically, an APPL/A relation declaration consists of a specification and a body. The specification for a simple APPL/A relation `Source_Repository` is shown in Figure 1. This relation stores source-code units, associating a module name and author name with the code for each unit. Features of APPL/A relations are explained below in terms of this example.

Each relation specification includes a defining tuple type that specifies the names and types of the attributes of the relation. A tuple type is similar to a record type, but tuple attributes have modes like Ada parameters. The attribute modes indicate the way in which attributes may take on values. Attributes of mode **in** must have values inserted directly by a user of the relation; this mode applies to all of the at-

tributes of `Source_Repository`. Attributes of mode **out** take on values that are automatically derived by the relation, whereas attributes of mode **in out** may take on given and derived values in turn. (Derived attributes are discussed below in reference to relation `Source_Compilations`.)

Each relation specification also includes a set of entries, analogous to Ada task entries, which represent the operations on a relation. The entries for a relation must be a non-empty subset of `insert`, `update`, `delete`, and `find`. The `insert` entry takes parameters for attributes of mode **in** and **in out** and implements the insertion of a tuple with those parameters into the relation. The `update` entry enables a tuple with given attribute values to be assigned new values for attributes of mode **in** and **in out**. The `delete` entry deletes a tuple with given attribute values. The `find` entry iteratively returns tuples selected by given attribute values.

The specification for another relation, `Source_Compilations`, is shown in Figure 2. This relation represents the derivation relationship between source code and the object code compiled from it. Unlike `Source_Repository`, this relation has some derived attributes, designated by mode **out**.

The specification of any relation with derived attributes may also contain a dependency specification, which indicates how the derived attributes are to be computed. In `Source_Compilations` the dependency specification states that, for each tuple, the values of the attributes `obj` and `messages` are to be computed by a call to the procedure `compile`, where the corresponding value of attribute `src` is taken as input. In this way `Source_Compilations` represents the derivation relationship established between the input and output of the `compile` tool. The body of the relation must carry out the computations necessary to assign values to derived attributes. If a relation has a dependency specification then the computation of attributes must be carried out according to that specification, and the computed values must be kept up-to-date with respect to the input values from which they are derived.

The body of a relation must generally implement the semantics of that relation. This means that the relation body must provide persistent storage, implement the relation entries, and compute and assign values for derived attributes. However, the details of the implementation can be left up to the programmer of the body (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 persistent storage system
- the derivation strategy for computed attributes (e.g. eager or lazy)

```
with Compile; -- separately defined compiler
with Code_Types; use 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 source_code;
        obj: out 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 2: Specification for Relation Source_Compilations

- the caching strategy for computed attributes (e.g., cached when computed or recomputed when needed)

The implementor of a relation can program the body in any way that satisfies the required semantics, and the implementation can even change over time without affecting users of the relation.

In providing a persistent data type, APPL/A can be regarded as a "persistent" programming language. Other such languages include PS-Algol [4], Adaplex [36] (which extends Ada with a functional data model), E [31] (the database implementation language of the EXODUS [6] extensible DBMS and an extension of C++), and Owl [34] (the object-oriented language of the Trellis environment). This is a diverse group of languages, and APPL/A differs from each of them in many particulars.

Support for Change Management APPL/A relations combine several capabilities that are recommended for change management in Section 2.5:

- They provide a data structure for the explicit representation of relationships among objects. Relations can be used to determine the direction and extent of propagation of changes to objects.
- They encapsulate derivation processes. Derivation dependencies are represented explicitly and maintained automatically. Thus relations free developers from the need to track and maintain derivations manually.
- They are abstract types with programmable implementations. Consequently, they serve to isolate logical from implementation issues. The implementation can be varied without affecting users of the abstract interface, and processes can be programmed in terms of the interface without regard for implementation details.

Each of these capabilities is illustrated or discussed in the example in Section 4. Their integration in relations makes relations especially useful in change management.

3.2 Triggers

An APPL/A trigger unit is like an Ada task unit in that it represents a concurrent thread of control. However, triggers differ from tasks in that triggers lack entries. Instead, triggers *react* indirectly and automatically to operations on relations.

A trigger has a simple specification, comparable to an Ada task's but without the entries. A trigger body comprises a loop over a selective trigger statement. A selective trigger statement is like an Ada selective wait statement, except that it has "upon" alternatives instead of "accept" alternatives. Each upon alternative consists of an upon statement followed by a (possibly empty) sequence of statements. The upon statements identify the relation operations to which a response is to be made. The statements within and immediately following the upon statement encode the trigger's response to the relation operation.

The body of a trigger `Maintain_Source_Compilations` is shown in Figure 3. The purpose of this trigger is to automatically assure that every module in `Source_Repository` is represented in `Source_Compilations`. The trigger responds to operations on relation `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`; analogous responses are made to update and delete operations.

Trigger `Maintain_Source_Compilations` includes three upon statements, one each for the insert, delete, and update entries of `Source_Repository`. Each of these upon statements is for a completion event, i.e. a response is to be triggered only upon the successful completion of the corresponding entry call. (Upon statements can also designate acceptance events, in which case a response would be triggered by the acceptance of the relation entry call.) Each upon statement also includes a list of formal parameters. For an acceptance event these comprise the **in** parameters for the relation entry call; for a completion event these comprise the **in**, **in out**, and **out** parameters for the call. Through these parameters the actual values given to and returned from the relation entry call are made available to the trigger. Although it is not shown in the example, upon declarations may also be given priority values. When an event occurs (i.e. a relation entry call is accepted or completed), a signal is sent to each trigger that designates that event in an upon statement. This signal includes the identity of the event and the corresponding actual parameters. Event signals are queued at the trigger in order of priority and responded to in turn.

It should be noted that a trigger can make both "synchronous" and "asynchronous" responses to events. The body of an upon statement (within the **do** ... **end** block) is executed synchronously with the event signal in the same sense that an accept statement is executed synchronously with an entry call. While the upon statement is executing, the execution of the triggering relation is suspended at the point at which the signal was generated (either acceptance or completion of the rendezvous for the relation entry). However, the trigger does not execute a full rendezvous with the relation, and no parameters or exceptions are returned from the trigger to the relation. Once the upon statement completes,

```
trigger body Maintain_Source_Compilations is
  sc_t: src_compilations_tuple;
begin
  loop
    select
      upon Source_Repository.insert(
        author: in name_type;
        name: in name_type;
        src: in source_code)
      completion do
        -- propagate name and source to
        -- Source_Compilations
        Source_Compilations.insert(name, src);
      end upon;
    or
      upon Source_Repository.update(
        author: in 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 and
            -- src in Source_Compilations
            ...
        end if;
      end upon;
    or
      upon Source_Repository.delete
        author: in name_type;
        name: in name_type;
        src: in source_code)
      completion do
        -- delete tuple with name and src
        -- from Source_Compilations
        ...
      end upon;
    or
      terminate;
    end select;
  end loop;
End Maintain_Source_Compilations;
```

Figure 3: Sketch of Trigger Body Maintain_Source_Compilations

the synchronization with the relation is released and the trigger and relation proceed in parallel. A sequence of statements immediately following an upon statement thus executes asynchronously with the relation and can be used to provide an asynchronous response to relation operations.

Support for Change Management Triggers support change management by automating responses to change. Triggers react to operations that change relations. They can be used to propagate data, send notifications, invoke tools, log changes, and perform other tasks. Thus triggers can assume many of the duties of change management that are left to humans in the scenario of Section 2. Because of their "reactivity", triggers can also be easily added to and deleted from process programs. These changes to programs can be made without affecting the relations to which the triggers respond (or which they call), thus facilitating process-program evolution. A trigger plays a central role in automating changes in the example of Section 4.

3.3 Predicates and Consistency

An APPL/A predicate unit allows the process programmer to specify conditions on relations and to indicate (optionally) whether they should be enforced like constraints. A predicate unit is a named boolean expression over relations. The expression language includes existentially and universally quantified forms and conditional expressions. Two predicates are shown in Figure 4. The first of these tests the uniqueness of name in `Source_Repository`; the second tests the integrity of name references between `Source_Compilations` and `Source_Repository`. Keywords used in these examples are explained below.

When a predicate in a program is enforced during the execution of that program, then no operation by the resulting process on the relations to which the predicate applies is allowed to terminate in violation of the predicate. Any such operation is undone and causes an exception to be raised. An enforced predicate thus acts like a constraint on the relations to which it applies. Unlike conventional constraints, however, the default enforcement of APPL/A predicates can be turned on and off dynamically. This adds a dimension of flexibility to consistency management in that it allows constraints to be imposed when they are considered important but to be relaxed at other times. (If a predicate is to be enforced at all times, then it may be declared **enforced**, in which case its default enforcement cannot be turned off.)

Predicates may be global or local. A global predicate is so designated by the keyword **mandatory**. The extent of a global predicate includes all programs which use relations to which the predicate refers. A local pred-

```
-- a predicate to test 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;

-- a predicate to test referential integrity
mandatory enforced 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;
```

Figure 4: Some Basic Predicates

icate may be included optionally in any program but it need not be included in any; its extent is restricted to those programs in which it is explicitly included. The availability of global and local predicates adds another dimension of flexibility to consistency management in that it allows specification of the scope in which a predicate should be enforced. Global predicates can thus serve as process-independent constraints, whereas local predicates can serve as process-dependent constraints.

Each predicate has a boolean Ada-style "attribute" `enforced` which indicates whether it is enforced by default. For a predicate declared **enforced** this attribute is a constant true. For other predicates this attribute is a variable which may be set to set the default enforcement of the predicate. APPL/A provides a capability mechanism to control assignment to predicate `enforced` attributes; this mechanism is illustrated in Figure 19 of Section 4.4. `Enforced` attributes are not the only mechanism by which predicate enforcement can be controlled, however. The default enforcement of a predicate, even a predicate declared **enforced**, can be locally overridden within certain transaction-like consistency-management statements, as described in the following section.

Support for Change Management Predicates facilitate change management in several important ways. Predicates allow the intended state of relations to be explicitly expressed, and the enforcement of predicates helps to assure that intended states are maintained. The use of predicates thus provides guidance in making changes to relations and helps to preclude inconsistent changes. The evolution of consistency is also supported in that predicates can be added to and deleted from process programs without affecting the relations to which they refer, and the enforcement of existing predicates can be turned on and off over time. This kind of evolution is essential in software processes and process programs, but it is not supported by conventional databases or many advanced object-management systems.

3.4 Consistency-Management Statements

The consistency-management statements serve to group individual relation operations into composite operations that have transaction-like properties. They include the suspend, enforce, allow, serial, and atomic statements. For brevity, these are referred to below as the "CM" statements.

The CM statements are designed to allow more flexibility than is afforded by conventional transactions and to support the construction of alternative "high-level" or "long-term" transactions that are required for software processes. Individually, the statements are more specialized than conventional transactions in terms of serializability, atomicity (rollback), and predicate enforcement. However, the statements can be nested, and they may include concurrent tasks, so they can be used to implement a wide variety of advanced transaction models, such as nested, concurrent transactions, hierarchical transactions, "assertion transactions", and more. Each of the CM statements is described below.

Serial Statement The serial statement provides simple serializable read or write access to relations. The serial statement includes a "read-write list" which identifies relations to which read or write access is desired. (Read access allows other readers but excludes writers, write access is all together exclusive.) Serial statements are used in the example of Section 4 in Figures 14, 16, and 19.

The serial statement does not affect the default or actual enforcement of predicates. Within a serial statement any operation that violates an enforced predicate is individually rolled back; there is no rollback for the statement as a whole.

Suspend Statement The suspend statement provides a context in which the actual enforcement of desig-

nated predicates is temporarily and locally suspended. The suspend statement provides serializable write access to the relations to which the suspended predicates apply. Write operations on these relations are logged. Upon completion of the suspend statement any of the suspended predicates which are enforced in the surrounding scope must be satisfied or the logged operations are rolled back. A suspend statement is shown in Figure 15 in Section 4.

Enforce Statement The enforce statement provides a context in which the actual enforcement of designated predicates is temporarily and locally imposed rather than suspended. Any operation within the scope of the enforce statement that violates an enforced predicate is individually undone as it occurs. Consequently, there is no need for rollback for the statement as a whole. Because there is no need to protect concurrent processes from rollback or consistency violations, the enforce statement is not serializable. However, if desired, serializable access to relations can be obtained by nesting an enforce statement within a serial statement. (Similarly, if serializable, recoverable access is desired, an enforce statement can be nested within an atomic statement.) The use of an enforce statement (within an atomic statement) is sketched in Figure 5 (explained below).

Atomic Statement The atomic statement provides serializable and recoverable access to relations. The atomic statement has a read-write list of relations to which serializable access is requested. Write operations on these relations are logged. The atomic statement does not affect predicate enforcement. However, the propagation of an exception from the atomic statement does cause rollback of the logged results of the statement. Because the atomic statement may entail rollback it is serializable. An atomic statement is used in Figure 5.

Allow Statement The allow statement also creates a context in which the enforcement of predicates is suspended. Any designated predicate that is violated upon entry to the allow may be violated by operations within the allow *and also upon exit from it.* This allows an *existing* predicate violation to be perpetuated and thereby admits the possibility of only partial repair of the violation. No other enforced predicates may be violated within or upon completion of the statement. The violation of any other predicate by an operation in the allow statement causes that particular operation to be undone (although a suspend statement can be nested within an allow statement to suspend the enforcement of other predicates). There is no rollback for the statement as a whole. However, because it suspends the enforcement of violated predicates, the allow statement is serializable with respect to operations on the relations referenced by those predicates.

A Simple Example Showing the Use of the Statements A simple example with atomic and enforce statements is shown in Figure 5. The construct shown might be called "assertion transaction." An enforce statement is used to create a scope in which the predicate `Source_Length_Within_Limit` is locally enforced. This predicate (not shown) tests whether source modules in `Source_Repository` conform to a maximum length limit. Within the enforce statement this predicate becomes an added assertion on consistency of the relation. If the predicate is violated, the offending operation is undone and an exception is raised. The enforce statement is nested within an atomic statement. The atomic statement does not affect predicate enforcement, but it locally provides serializable and recoverable write access to `Source_Repository`. If an exception is propagated from the enforce statement (for any reason) it will cause roll back of the atomic statement and, hence, of the nested enforce statement. In this way, an effect similar to a conventional transaction is achieved, but with a strengthening rather than a relaxation of consistency requirements.

```
atomic write Source_Repository;
begin
    enforce Source_Length_Within_Limit;
    begin
        -- operate on Source_Repository here,
        -- constrained by the Source_Length_-
        -- Within_Limit; violation of the
        -- predicate will cause an exception
        -- to be raised.
        ...
    end enforce;
end atomic; -- rollback upon exception
```

Figure 5: Sketch of a simple "Assertion Transaction"

The example shows that the CM statements can be nested. Rules for nesting and concurrency within these statements are essentially those defined by Moss [24] for nested, concurrent transactions. It is also possible in APPL/A to have nested transactions which are functionally separate from their nesting transactions. Rules for combination of the statements, the rationale for their design, and additional examples are found in [39, 40].

Support for Change Management The consistency-management statements support change management by enabling a process to coordinate access to data

with other processes and to establish a local regime for concurrency control, atomicity, and predicate enforcement. Thus these statements help to assure that changes are made consistently, atomically, and free from interference or observation by outside processes. The statements enable a process to protect itself from and adapt itself to outside changes in the enforcement of predicates and the consistency of relations. In the example of Section 4, the CM statements play an important role in assuring that changes are made correctly and in evolving existing data to satisfy newly-imposed constraints.

3.5 Additional Comments on APPL/A Support for Change Management

The constructs in APPL/A support change management in several ways, as discussed above for each kind of construct. These constructs as a group allow for the explicit representation of relationships, constraints, and processes (including transactions) that are essential to change management. They can be used in programs that automate software processes, including various kinds of change. Concomitantly they reduce the reliance on humans to understand and manage change manually. In light of these observations and the observations presented in preceding subsection, we believe that the constructs introduced in APPL/A generally meet the recommendations presented in Section 2.5. Thus they should generally facilitate change management in process programs.

4 An Extended Example

This section presents an extended example of change management based on a process program written in APPL/A. The process defined by the program includes the coding and compiling of a set of modules, with associated reviews. The example is presented in four parts, each treated in a subsection below. Each subsection describes relevant aspects of the program and concludes with a discussion of issues related to change management. Section 4.1 presents the basic process program. This illustrates change management within a software process, as represented and implemented by the program. Section 4.2 shows how a simple change to the process is made through a simple change to the program; Section 4.3 shows a more complicated change to those same parts of the process and program. Finally, Section 4.4 presents a program which facilitates change between the later versions of the process. Several other points about process programs are also illustrated through these examples.

4.1 The Initial Program: Code_and_Compile

The process program is referred to below after its main procedure, `Code_and_Compile`. `Code_and_Compile` illustrates many APPL/A features and shows how they can be used to support change management in a software process. Section 4.1.1 gives an overview of the process and program. Section 4.1.2 introduces the APPL/A relations used in the program, and Section 4.1.3 indicates some of the associated APPL/A predicates. Section 4.1.4 then presents procedures, triggers, and tasks that exemplify the main control elements of the program. As described below, the execution of the program is driven more by a trigger than by the main procedure. That is because the emphasis in this example is on change management, and triggers are especially appropriate for reacting to changes in relations and invoking an appropriate responses. A concluding discussion of change-management issues is presented in Section 4.1.5.

4.1.1 Overview

The process that is programmed in `Code_and_Compile` consists of four main activities: coding of source modules, review of source modules, compilation of source modules, and review of compilation results. Source modules are reviewed to determine their readiness for compilation; if a module is accepted it is compiled, otherwise it is recoded. Compilation results are evaluated to determine the success of compilation; if the results are accepted the module is considered complete, otherwise it is recoded. The process terminates when all modules have been successfully compiled.

The process combines both manual and automated activities, doing so in various ways. Coding, which is a manual activity, is accomplished by notifying the person responsible for the task and then awaiting the result of their actions, i.e. the source-code module, which is accepted as input. Compilation is automated and is accomplished by encapsulating the compiler in a relation, where it is automatically invoked as needed. Reviews may be implemented by manual inspection and/or automated analysis; changes to the implementation of the code review provide the basis for the examples in following sections.

An overview of the main program units of `Code_and_Compile` is presented in Figure 6. This figure shows many of the control units, relations, and predicates which are discussed in this section and indicates several of the important relationships among them. To reduce the complexity and length of the example, several aspects of the process that are less relevant to change mangement have been simplified or abstracted away. Many procedures are represented by specification only; also omitted are the bodies of relations. Context

clauses are left out where they can be reasonably inferred. Exception handling is largely ignored, as are provisions for gracefully terminating the process short of completion. Additionally it is assumed that certain data will not change during the process. These include the relation `System_Structure` and the names of modules (which are used as unique identifiers and relation keys). Further details on the process and on the design of `Code_and_Compile` are provided below where relevant to the presentation of the code.

4.1.2 Relations

The program `Code_and_Compile` includes several relations:

System_Structure This is a binary relation (not shown) that relates the names of executable systems to the names of the modules from which they are composed. It identifies the units to be coded and compiled. (The linking of executable systems is beyond the scope of this example. However, `System_Structure` would be used in a linking process to identify the object modules required for a system build and to retrieve those from relation `Source_Compilations` (described below). Thus this relation exemplifies the way in which relationships among data shared by different processes may be represented.)

Source_Repository This relation was shown in Figure 1. It is used to hold the baseline copies of source modules. When a source-code unit is created or revised it is entered here prior to evaluation, compilation, and so on. Module name serves as a unique identifier for modules. Author name is not used in the process but is included to suggest that additional information may be associated with modules (as it would be in a more realistically detailed process).

Source_Compilations This relation was shown in Figure 2. It represents the derivation relationship between source modules and the object modules compiled from them. Through this relation the dependence of object code on the source code from which it is compiled is made explicit, the required compiler is encapsulated, and the compiler is automatically invoked as necessary to keep the object code up-to-date with the source code.

Module_Status This relation maintains information about the coding and compilation status of each module. A sketch of the relation is shown in Figure 7.

Status values are defined by an enumerated type:

```
type status_value = (incomplete, unevaluated,
      rejected, accepted, outdated);
```

```
Relation Module_Status is
-- Stores values to represent the status of
-- coding and compiling for modules.
--
    type module_status_tuple is tuple
        name: name_type;
        code_status, compile_status:
            status_value;
    end tuple;
entries
    ... -- standard insert, update, delete, find
End Module_Status;
```

Figure 7: Sketch of Specification for Relation Module_Status

The intended semantics of code status values are as follows:

- `Incomplete`: The source code for the module is not represented in `Source_Repository`, which is intended to hold the baseline for all source code.
- `Unevaluated`: The source exists in `Source_Repository` but has not been explicitly evaluated to determine whether it is ready for compilation or must be recoded.
- `Accepted`, `Rejected`: The source code has been evaluated with regard to its readiness for compilation. `Accepted` implies that the source is ready for compilation. `Rejected` implies that the source should be revised; either it is not ready for compilation or compilation has failed.
- `Outdated`: The module exists but is being revised.

Values for compilation status have analogous meanings.

Maintaining the consistency of module status values is an important issue in this process program. For example, a module's compile status should not be `accepted` while its code status is `rejected`. Constraints such as these are specified by separately declared predicates, discussed in the next section.

4.1.3 Predicates

This section presents some of the predicates that are used to define and enforce the consistency of relations in `Code_and_Compile`.

One important type of consistency is the uniqueness of attribute values and tuples in relations, i.e., key constraints. For example, module name is intended to serve as a key in the relations `Source_Repository`, `Source_Compilations`, and `Module_Status`; consequently module names should be constrained to

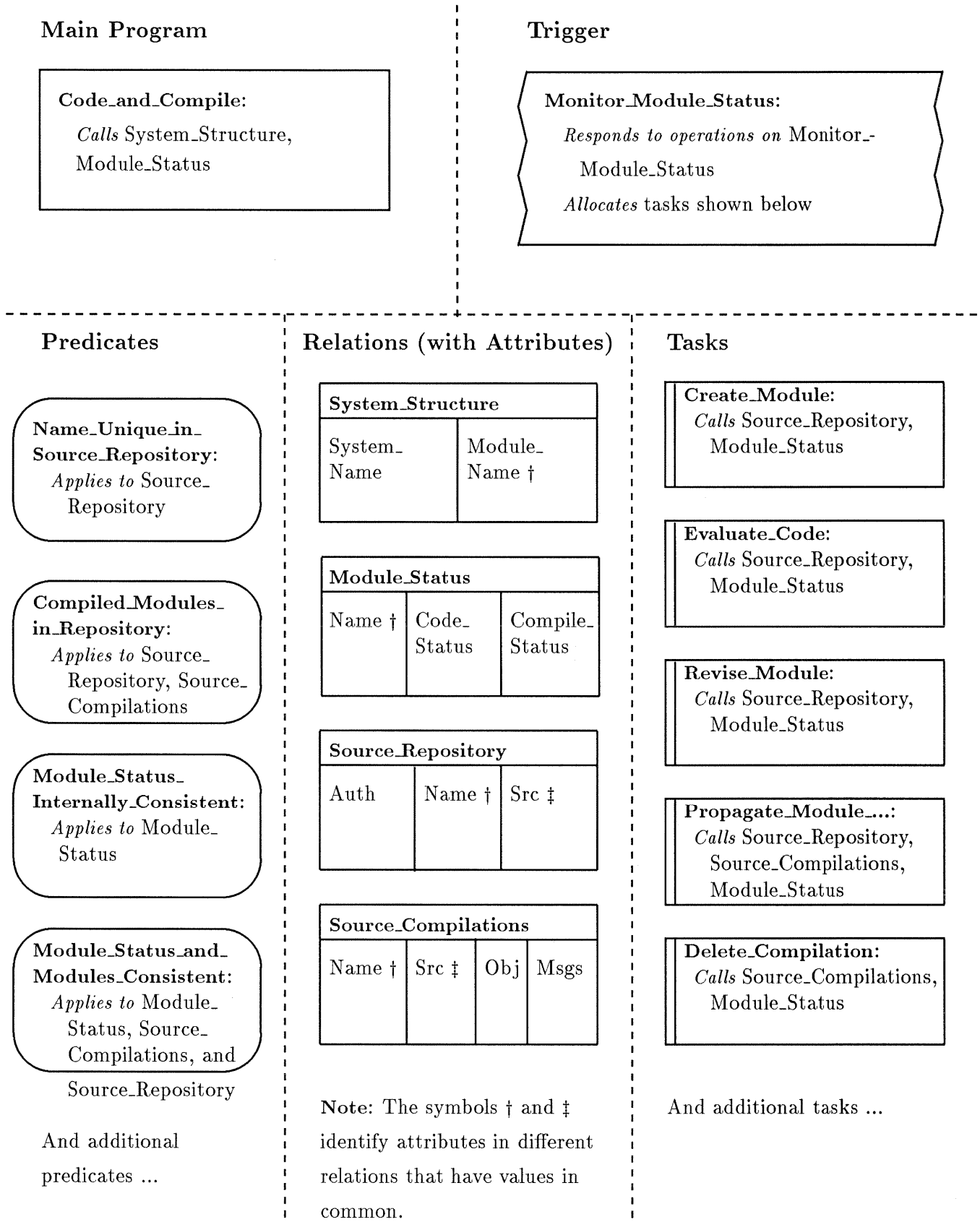


Figure 6: Principal Units in the Program `Code_and_Compile`

```
mandatory predicate
  Module_Status_and_Modules_Consistent is
begin return
  every t1 in Module_Status
  satisfies
    if t1.code_status = incomplete
    then
      no t2 in Source_Repository
      satisfies
        t1.name = t2.name
      end no
    else -- t1.code_status /= incomplete
      some t2 in Source_Repository
      satisfies
        t1.name = t2.name;
      end some
    end if
  and   -- similarly for compile status
      ... -- and Source_Compilations
  end every;
End Module_Status_and_Modules_Consistent;
```

Figure 8: A Predicate to Test the Consistency of Modules and their Status

```
mandatory predicate
  Module_Status_Internally_Consistent is
begin return
  every t in Module_Status satisfies
    if t.compile_status in
      unevaluated..accepted
    then
      t.code_status = accepted
    else
      true
    end if
  and
    if t.code_status /= accepted then
      t.compile_status = incomplete or
      t.compile_status = outdated
    else
      true
    end if
  end every;
End Module_Status_Internally_Consistent;
```

Figure 9: A Predicate to Test the Internal Consistency of Module Status Values

be unique in those relations. An example of a predicate which enforces such a condition is shown in Figure 4.

Another important type of consistency is inclusion dependencies between relations (i.e., referential integrity). This condition applies when attribute values in one relation should be a subset of those found in another relation. For example, `System_Structure` is taken to define the set of legitimate module names for use in `Code_and_Compile`. Consequently, only those names should appear in the module-name attributes of the other relations in the program. Similarly, the modules represented in `Source_Compilations` may be limited to a subset of those in `Source_Repository`. This condition would be enforced if it were considered desirable to keep both copies of a source module consistent and up-to-date. However, consistency between separate copies of source modules is not necessarily a requirement of all coding processes that might use these relations. An out-of-date version of a source module and the corresponding object code may be allowed to remain in `Source_Compilations` when that version of the source code has been deleted from `Source_Repository` in anticipation of an updated version. This illustrates that choices about which predicates to enforce as constraints, and where and when they should be enforced, are process-dependent design decisions. A predicate that tests inclusion dependencies is shown in Figure 4.

Other predicates are used to define other kinds of consistent states within and between relations. In `Code_and_Compile` the consistency of status values is important. For a module, the compile status should not be greater than the coding status, since the quality of compilation results depends on the quality of the code compiled. So, for example, the compilation status for a module should not be `accepted` while the coding status for the module is `rejected`, and if the compilation status for a module is `accepted` or `rejected` (implying that the module has been compiled) then the coding status for the module should be `accepted` (implying that the module is available for compilation).

Another aspect of consistency related to the status values is the correspondence of status values with the objects to which they refer. For example, a module code status value of `incomplete` is intended to imply that the module is not represented in `Source_Repository`. Some examples of predicates governing the consistency of status values are shown in Figures 8 and 9.

4.1.4 Procedures, Tasks, and Triggers

The procedure `Code_and_Compile` (Figure 10) represents the main program for the process. The body of this procedure is a simple loop to initialize the code and compile status for each module to `incomplete`. All other operations of the process are performed reactively by the trigger `Monitor_Module_Status`, which allocates tasks as necessary to accomplish the concurrent creation, propagation, and evaluation of data.

Trigger `Monitor_Module_Status` effectively con-

```
procedure Code_and_Compile is
-- Main program of code and compile process.
    trigger Monitor_Module_Status;
    trigger body Monitor_Module_Status
        is separate;
Begin
    -- Initialize module status
    for ss_t in System_Structure loop
        if not Module_Status'member(
            select_name => true,
            name => ss_t.module_name)
        then
            Module_Status.insert(
                mn_t.name, incomplete,
                incomplete);
        end if;
    end loop;
    -- Subsequent operations performed auto-
    -- matically and concurrently by trigger
    -- Monitor_Module_Status and tasks that
    -- it allocates
End Code_and_Compile;
```

Figure 10: Procedure Code_and_Compile

trols the execution of the program following the initialization of status values. It responds to insertions or updates of module-status values and invokes whatever operation is called for by the new status. These operations characteristically involve the allocation of a task to carry out required work concurrently. In this way the trigger acts as an interpreter of module status. This interpretation continues until all modules have a compile status of `accepted`, at which point the trigger, and consequently the main procedure, terminate. A part of this pattern of execution is illustrated in Figure 4.1.4. The body of the trigger is shown in Figure 12.

`Monitor_Module_Status` makes use of two main subprocedures, one to respond to changes in code status, and one to respond to changes in module status. Each of these encapsulates a case statement in which the appropriate response to each status value, if any, is invoked. The procedure `Respond_to_Code_Status` (Figure 13) makes the following responses:

- If the new value is `incomplete` then the procedure `Create_Module` is invoked to (re)create the named module.
- If the new value is `unevaluated`, the procedure `Evaluate_Code` is invoked to evaluate the source code for the module. The evaluation will return a recommendation to accept or reject the source, and the module's code status is then set accordingly.

```
trigger body Monitor_Module_Status is
-- Respond to changes of status values
-- in Module_Status
  ...
Begin
  loop
    select
      upon Module_Status.insert(
        name: name_type;
        code_status, compile_status:
            status_value)
      completion do
        respond_to_code_status(
          name, code_status);
        respond_to_compile_status(
            name, compile_status);
      end upon;
    or
      upon Module_Status.update(
        name: name_type;
        code_status, compile_status:
            status_value;
        update_name: boolean;
          new_name: name_type;
        update_code_status: boolean;
          new_code_status: status_value;
        update_compile_status: boolean;
          new_compile_status: status_value)
      completion do
        if update_code_status then
          respond_to_code_status(
            name, new_code_status);
        end if;
        if update_compile_status then
          if new_compile_status = accepted
              and then
              all_compile_status_accepted
          then
            exit;
          else
            respond_to_compile_status(
              name, new_compile_status);
          end if;
        end if;
      end upon;
    or
      terminate;
    end select;
  end loop;
End Monitor_Module_Status;
```

Figure 12: Trigger Body Monitor_Module_Status

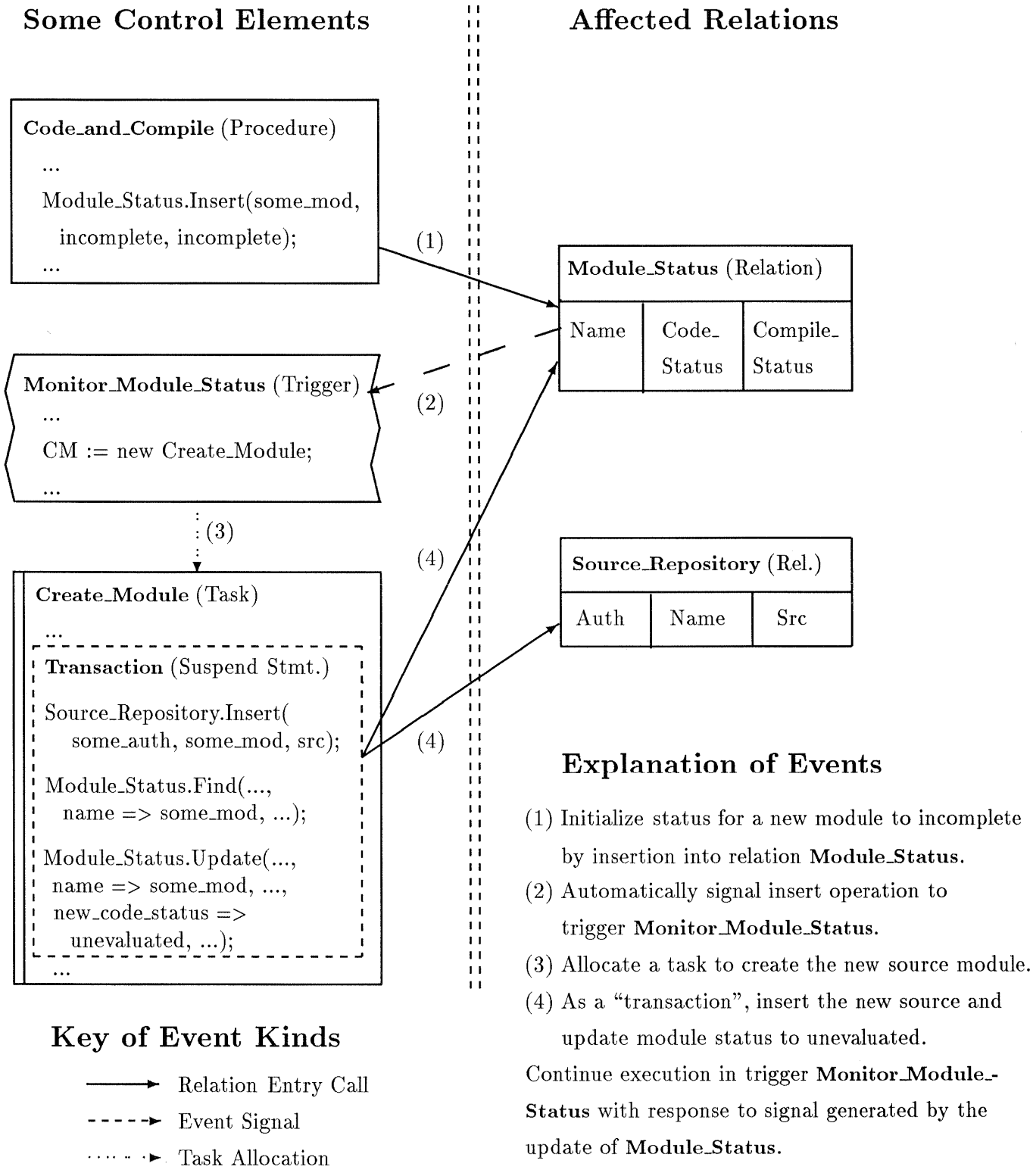


Figure 11: A Characteristic Sequence of Events in the Execution of `Code_and_Compile`

- If the new value is **accepted** then the procedure **Propagate_Module_for_Compilation** is invoked to insert the name and source code for the module into relation **Source_Compilations**. The trigger thus automates the propagation of the source code between relations. The inclusion of source code in **Source_Compilations** also assures that the code will be automatically compiled.

- If the new value is **rejected** then the procedure **Revise_Module** is invoked. This procedure sets the code status for the module to **unevaluated** and allocates a task to revise the source code. When that task subsequently updates the source code in **Source_Repository**, the module code status is set to **unevaluated**.

- If the new value is **outdated** then the corresponding tuple from **Source_Compilations** is deleted. This implements a policy that "dangling" source and object code should not be left in **Source_Compilations** once the base module (in **Source_Repository**) has been deleted.

The procedure **Respond_to_Compile_Status** has a similar structure. When a module is newly compiled, it invokes the procedure **Evaluate_Compilation**; when a compilation is rejected, it invokes the procedure **Revise_Module**. Otherwise, it takes no action: if the new status is **accepted** the module is considered finished, whereas if the new status is **incomplete** or **outdated** the insertion or update of source code must occur before any other response.

The procedures for the various actions taken in response to status changes are declared in a package **Code_and_Compile_Subroutines**. A sketch of the body of this package, showing some of the procedures, is found in Figure 14.

The procedure **Create_Module** can serve as a model of the subroutines used in responding to new status values. The role of the procedure is to allocate and initiate a task of type **Create_Module_Task** (Figure 15), which represents the real work in coding a new module.

The task is passed the name of the module to be coded, and it then requests the identifier of the programmer who is assigned to code the module. The task notifies the programmer that the module is to be coded, and then it waits for the code to be returned. The coding activity is presumed to be performed more or less manually (as may be the process of assigning the programmer in the first place). Thus this task illustrates how the interaction of manual and automated activities can be defined in an APPL/A process program.

When the programmer returns the source code for the module the code is inserted into **Source_Repository** and a code status of **unevaluated** is assigned to the module in **Module_Status**. All of these updates are

```
procedure Respond_to_Code_Status(
    module: in name_type;
    status: in status_value) is
Begin
    case status is
        when accepted =>
            propagate_module_for_compilation(
                module);
        when rejected =>
            revise_module(module);
        when incomplete =>
            create_module(module);
        when unevaluated =>
            evaluate_code(module);
        when outdated =>
            delete_compilation(module);
    end case;
End Respond_to_Code_Status;
```

Figure 13: Procedures to Respond to Changes in Module Code and Compile Status

made within a suspend statement. The suspend statement provides serializability and atomicity for the composite update, thus assuring that intermediate states during the update will not be accessible to outside processes. It also locally suspends the enforcement of the predicate **Module_and_Module_Status_Consistent**, which is violated by the update of **Source_Repository**. Within the suspend statement that violation is allowed to stand temporarily, and it is subsequently repaired by the update of **Module_Status**. The statement as a whole then leaves the relations in a new state which does satisfy the predicate. If the update to **Module_Status** were to fail for some reason, then the predicate would not be satisfied. In that case, though, the suspend statement would roll back, returning the relations to the state which held prior to the execution of the statement. In that way the consistency of relations is preserved.

The rationale for the use of a separate task in **Create_Module** bears noting. The allocation of a separate task helps to increase concurrency and also reduces the potential for deadlock among the various relations and triggers. Such allocated tasks are free to call whatever relations are necessary without blocking the triggers that allocated them; those triggers are free to respond to further operations on relations, including operations performed directly or indirectly through the allocated tasks.

Some comments should also be made about the other procedures invoked (indirectly) by trigger **Monitor_Module_Status**. Procedure **Revise_Module** (not shown) is substantially similar to **Initiate_**

```
package body Code_and_Compile_Subroutines is
-- Procedures used by Respond_to_Code_Status
-- and Respond_to_Compile_Status in responding
-- to changes of module status values.

    procedure Create_Module(
        module: in name_type) is
    -- Allocate a task for concurrent coding
    -- of the named module
    --
        task type Create_Module_Task is
            entry initiate(
                module_name: in name_type);
        end Create_Module_Task;
        ...  -- other declarations
        task body Create_Module_Task
            is separate;
    Begin
        ...  -- allocate and initiate a
             -- Create_Module_Task
    End Create_Module;

    procedure Evaluate_Code(
        module: in name_type) is
    -- Set module code status to "accepted"
    -- on the assumption that it has been
    -- adequately inspected by the programmer
    Begin
        serial read Module_Status;
            ms_t: module_status_tuple;
        begin
            ms_t := Module_Status'tuple(
                select_name => true,
                name => module);
            Module_Status.update(ms_t.name,
                ms_t.code_status,
                ms_t.compile_status,
                update_code_status => true,
                new_code_status => accepted);
        end serial;
    End Evaluate_Code;

    ...  -- other procedures
End Code_and_Compile_Subroutines;
```

Figure 14: Body of Package `Code_and_Compile_Subroutines`

```
task body Create_Module_Task is
-- A task for the initial coding of a module.
    module: name_type;
    programmer_id: id_type;
    code: source_code;
    reply_id: id_type := new_id;
    sr_t: proj_src_tuple;
    ms_t: module_status_tuple;

    procedure Get_Programmer_Id(
        module: in name_type;
        programmer: out id_type)
    is separate;
Begin
    -- get module name and programmer id
    accept initiate(module_name: in name_type)
    do
        module := name_type;
    end accept;
    get_programmer_id(module, programmer_id);

    -- notify programmer of module to create
    -- and await new code
    send_message(programmer_id, reply_id,
        "Create module: " & module);
    recv_message(reply_id, programmer_id,
        code);

    -- save new code and update code status
    suspend
        Module_Status_and_Modules_Consistent;
    begin
        -- insert new code
        Source_Repository.insert(
            programmer(programmer_id),
            module, src);
        -- set module code status to "unevaluated"
        ms_t := Module_Status'tuple(
            select_name => true,
            name => module);
        Module_Status.update(
            ms_t.name, ms_t.code_status,
            ms_t.compile_status,
            update_code_status => true,
            new_code_status => unevaluated);
    end suspend;
End Create_Module_Task;
```

Figure 15: Task Body Create_Module_Task

Coding. It represents the revision of an existing module rather than the creation of a new module. When the revised source code is submitted by a programmer, the task updates Project_Source_Repository to replace the old value of the source code with the new value. (Status values are also atomically and consistently updated as in Code_Module_Task.) Thus, Revise_Module reflects a simple policy regarding the management of outdated source-code modules. The particular process is not so important here; what is important is that the policy is represented explicitly and implemented automatically.

Code_and_Compile also includes subprocedures to evaluate the results of coding and compiling. The procedure to evaluate the results of coding (shown in Figure 14) is trivial in that module code status is always set to accepted on the assumption that the programmer only submits the code once it is of acceptable quality. The procedure to evaluate the result of compilation is more complicated in that it may recommend either acceptance or rejection. The details of this procedure are not spelled out here. Presumably the recommendation depends on some sort of examination of the messages generated by the tool involved. In general that examination may be manual or automatic. For example, compilation may be rejected automatically if any messages are generated; alternatively, it may be rejected automatically if error messages are produced, but turned over for manual review if only warning messages are produced. The evaluation procedures thus have two aspects which are interesting from the perspective of process programming: they represent the encapsulation of development policies, and they represent the opportunity for the integration of manual and automatic activities in the process.

4.1.5 Discussion of Change Within the Process

The process program Code_and_Compile supports the management of change in several ways recommended in Section 2.5. It provides an explicit representation of many aspects of the process, including those parts involved with change. It makes many kinds of change automatically, for example, the propagation of code between relations and the corresponding adjustments of status. It provides automated control over manual changes: the activities that require manual participation are automatically invoked, and the products of manual activities are automatically inserted into the proper relations. The combination of automated and manual changes is also automatically coordinated, so that, for example, when a programmer submits a new module, the status for the module is also updated.

Relations play several roles with respect to change management. They represent relationships between objects, and they can be used to determine the direction and extent of dependent changes that follow when an object is changed. For example, when a source module is updated in Source_Repository, related status values can be found and updated in Module_Status, and dependent object modules can be found and updated in Source_Compilations. The relation System_Structure could similarly be used to track down executable systems that depend on a modified source module. The relation Source_Compilations not only represents the derivation relationship between source code and object code, it also encapsulates the compiler and automatically invokes it as necessary to keep the object code up-to-date with changes to the corresponding source code.

The trigger Monitor_Module_Status plays an important role in automating change management. In response to changes in Module_Status it propagates updates, notifies users of tasks to be done, and invokes subroutines to accomplish a variety of tasks. Through these activities it becomes a principal controlling agent of the program.

Predicates are used to specify consistent states of relations and are enforced to assure that changes to relations are made consistently. For example, the enforced predicated Name_Unique_in_Source_Repository assures that Source_Repository cannot be updated so that two tuples have the same value for the name attribute. This prevents changes to module names which would invalidate the use of names as unique identifiers. The predicate Module_Status_and_Modules_Consistent (Figure 8) is enforced to assure a certain degree of consistency between modules and their status. It also helps to give observers of status values confidence that changes in status reflect changes in the state of modules.

Composite updates to relations can be made serializably, atomically, and consistently with the consistency-management statements. For example, the suspend statement is used to suspend enforcement of predicates over module status when changes to modules imply changes to module status. These cannot be performed in a single operation, but all must be accomplished if a correct and consistent state transition is to be achieved. The serial statement is used to provide concurrent (and possibly competing) tasks with serializable access to the data they need, thus preventing changes made by one process from interfering with those made by another.

4.2 A Simple Revision to the Process and Program

This section illustrates how a simple change to the process can be implemented by a small change to the process program. It involves a revision of the procedure Evaluate_Code (the original version of which is shown in Figure 14).

```
procedure Evaluate_Code(module:
    in name_type) is
-- Send module to manager for review
-- to determine status.
    new_status: status_value;
    mgr_id: id_type := get_manager_id;
Begin
    send_message(mgr_id, reply_id,
        "Review module: " & module);
    recv_message(reply_id, mgr_id,
        new_status);

    serial read Module_Status;
        ms_t: module_status_tuple;
    begin
        ms_t := Module_Status'tuple(
            select_name => true,
            name => module);
        Module_Status.update(ms_t.name,
            ms_t.code_status,
            ms_t.compile_status,
            update_code_status => true,
            new_code_status => new_status);
    end serial;
End Evaluate_Code;
```

Figure 16: First Revision of Evaluate Code

4.2.1 A Revised Procedure Evaluate_Code

To motivate the revision, suppose that the execution of `Code_and_Compile` leads to too many errors in the compilation of source modules. Suppose that a review of the process indicates that programmers do not consistently evaluate their own code adequately and that they too often submit it for compilation before it is likely to compile. Suppose further that, to address this problem, it is decided that source code should instead be evaluated by a manager and that the code status of each module should be set according to this evaluation. That change to the process can be implemented just by making a change to the procedure `Evaluate_Code`. Previously, this procedure simply set the code status of the given module to `accepted`. In the new version, it signals the manager to review the source code and receives back a status value which is assigned to the module. The revised procedure is shown in Figure 16.

4.2.2 Discussion of Change to the Process and Program

This part of the example illustrates that changes to a process can be made explicit by changes to its program. As with the initial version of `Evaluate_Code`, the new and modified activities of the revised version are automatically controlled and coordinated with other activities in the process. Updates to code status are still made serializably and consistently.

In this particular example the change to the program was limited to just one procedure. Changes may be more complex in general, but, as with conventional programs, proper modularization can limit the extent of revisions to process programs. Because APPL/A is an extension of Ada, even a change to the program requires some recompilation and relinking. As in this example, though, recompilation need not be extensive. Moreover, the time to recompile and relink the new program is likely to be small compared to the duration of the process it represents. Even coding of the revised program may be a comparatively brief activity.

The costs of recoding, recompiling, and relinking are not the only ones that may be associated with the revision of a process program. One potential cost is the loss of data in going from one version of the program to the next. In APPL/A, however, the data that are stored in relations are persistent, and this persistence enables the data developed using one version of a program to be carried over to a later version in which the relations are reused.

Another possible cost is the loss of work being performed manually. This loss may occur when the execution of a process program is terminated before completion. In this case, though, manual activities may be continued if their results can be used in the revised program (and if that program is prepared to accept them). For example, if `Code_and_Compile` is killed while in execution, the programmers who are coding modules may continue their work if it is expected that the modules will be useful in the revised program. Note that the costs indicated above are not peculiar to programmed processes; they may occur whenever a development process is changed. Process programming simply provides a context in which the costs can be seen, analyzed, and addressed explicitly.

The issues discussed above arise when the code of a process program is changed. The primary motivation for changing the code of a process program is to modify its behavior. An alternative approach to program modification in a compiled language such as APPL/A is to make the program interpretive based on certain data and then to change those data. Some interpretation is performed in `Code_and_Compile` in that changes to status values drive subsequent activities. Status values are changed, however, only as a part of the normal pattern of operation of the program. At the other end of the spectrum, the pattern of execution of a program can be made dependent on data that are not changed during ordinary execution of the process but that are changed only to change the behavior of the program. In

`Code_and_Compile` this could be implemented by a flag which determines the choice of alternative procedures for evaluating source code, where that flag was set by a manager either within or outside the scope of the process. The program would then represent, in effect, a parameterized process. In such cases, however, it may be argued that the "real" process is not just the pattern of behavior which results from the interpretation selected at any one time, but rather that the process is the whole interpretive scheme with all of its possible alternatives. In any case, the design of process programs with an interpretive element represents yet another approach to the mangement of change of in software processes, one which depends on a change of data rather than a change of code.

4.3 A More Complicated Revision to the Process and Program

This section illustrates a change to the process and the program that extends the previous changes. For purposes of this part of the example, suppose that the change to procedure `Evaluate_Code` does not adequately reduce compilation errors, and suppose that the remaining errors are found to be largely attributable to problems with inter-module dependencies, which managers do not effectively identify in their review of individual modules.

In order to address this problem, it is decided that two further requirements should be placed on the overall development process. First, allowed inter-module dependencies must be declared in advance of coding. Second, the criteria for acceptance of module source code should be strengthened by requiring conformance to the declared intermodule dependencies. The specification of allowed intermodule dependencies is beyond the scope of `Code_and_Compile`, but we shall assume that this information is available when `Code_and_Compile` is executed.

The process revision sketched above is implemented by the following changes to `Code_and_Compile`:

- The inclusion of relations to represent allowed and actual intermodule dependencies;
- The addition of predicates and related functions to specify and enforce the consistency of these relations; and
- Further revision of `Evaluate_Code` to take intermodule dependencies into account.

These changes are discussed below.

4.3.1 Relations

This revision of `Code_and_Compile` introduces two new relations that are instances of the same relation type.

```
relation type Binary_Name_Relation is
-- Represents a directed binary relation
-- between two named entities.
--
    type binary_name_tuple is tuple
        from, to: name_type;
    end tuple;
entries
    ... -- standard insert, delete, update, find
End Binary_Name_Relation;


--
-- Two instances of Binary_Name_Relation
-- to characterize module use
--
Allowed_Dependency: Binary_Name_Relation;
-- Represents the allowed "uses" relation
-- between modules designated by name: the
-- "from" module is allowed to use the
-- "to" module.
subtype allowed_dependency_tuple is
    Binary_Name_Relation.binary_name_tuple;

Actual_Dependency: Binary_Name_Relation;
-- Represents the actual "uses" relations
-- between modules designated by name: the
-- "from" module actually uses the "to"
-- module.
--
subtype actual_dependency_tuple is
    Binary_Name_Relation.binary_name_tuple;
```

Figure 17: Relations Representing Actual and Allowed Use Relationships among Modules

The relevant declarations are shown in Figure 17. The relation type is `Binary_Name_Relation`. It is a stored relation of tuples with two attributes, `from` and `to`, both of which have type `name_type`. This relation type is designed to represent sets of ordered (or directed) pairs of named objects. The two relation instances are `Allowed_Dependency` and `Actual_Dependency`, in which the names refer to modules. These relations are intended to represent the allowed and actual use by one module (the "from" module) of another module (the "to" module). They support a very simple and general kind of interface control, which is intended only to be suggestive of the kinds of precise interface control which are presented in [46]. As described below, these relations are used in evaluating modules to determine their readiness for compilation, and predicates over these relations are used to constrain the set of acceptable modules. As noted in the introduction to this section, the

relation `Allowed_Dependency` is produced by a design activity which occurs prior to execution of `Code_and_Compile`, but the relation `Actual_Dependency` is determined as part of `Code_and_Compile` as modules are coded.

```
package Dependence_Consistency is
--
  function Dependencies_Consistent(
      module: name_type) return boolean is
  Begin
    for t in Actual_Dependency loop
      if t.from = module or t.to = module
      then
        if not Allowed_Dependency'member(t)
        then
          return false;
        end if;
      end if;
    end loop;
    return true;
  End Dependencies_Consistent;

  mandatory predicate
    Accepted_Code_Dependency_Conforms
  begin return
    every ms_t in Module_Status satisfies
      if ms_t.code_status = accepted then
        dependencies_consistent(ms_t.name)
      else
        true
      end if
    end every
  End Accepted_Code_Dependency_Conforms;
--
End Dependence_Consistency;
```

Figure 18: A Function and Predicate for Consistency of Module Use

4.3.2 Predicates

Predicates introduced in the second revision of `Code_and_Compile` govern the consistency of the new relations individually and with respect to relations previously in the program. The most important of the new predicates is `Accepted_Code_Dependency_Conforms` (Figure 18). This predicate establishes a new criterion of consistency for modules that have a code status of `accepted`, one which requires their actual intermodule dependencies to be a subset of their allowed intermodule dependencies. The effect of this predicate is to impose a more stringent standard for modules that are compiled.

4.3.3 Second Revision of Evaluate_Code

The final piece of this revision to `Code_and_Compile` is another revision of procedure `Evaluate_Code`. In this revision (not shown) the actual external references of the module are determined by analysis and stored in relation `Actual_Dependency`. Then a check is made to see whether the `Actual_Dependency` tuples for the module are a subset of the `Allowed_Dependency` tuples for the module. If so, the module code status is set to `accepted`, otherwise, it is set to `rejected`.

4.3.4 Further Discussion of Change to the Process and Program

This example illustrates a more extensive and complicated change than the simple replacement of a procedure. However, because the original program was well modularized, the impact on existing code was still minor.

The new approach to evaluating source code that is represented here shows one way that new tools can be integrated into a process. In this case the tool is a data-dependency analyzer which is encapsulated in a new procedure; new tools may also be encapsulated in relations and triggers. Because this tool is represented explicitly in the program, its relationship to other activities (manual and automated) can be determined by analysis of the program. Additionally, the invocation of the tool is automatic.

Another feature of the change made here is the addition of a new predicate. This represents a change in the consistency requirements for the process, in particular a strengthing of constraints on the status of source-code modules. The new predicate is represented explicitly and enforced automatically. A potential problem, though, if the change described here is made dynamically while the process is executing, is that the existing source-module status values may not satisfy the new predicate. Violations of the predicate would then have to be repaired before the revised process could be successfully executed. That task is addressed in the next section.

4.4 A Dynamic Transition Between the Processes

As a final example of change management, suppose that the changes described above are to be instituted dynamically during the execution of a coding process. The intention is to kill the initial process program but to have the revised process program pick up where the initial one left off, in effect as if it had been executing all along. For this effect to be achieved, the objects produced by the initial process will have to be bound into the revised process. That will happen automatically

since the revised program makes use of the relations from the initial program and the data in those relations are persistent. However, the code units in those relations may have status values that are inconsistent with the new constraints on module use. In order to adapt the existing code to the new constraints it is necessary to determine the intermodule dependencies for the existing modules, check those against the allowed intermodule dependencies (assuming these have been specified), and adjust the status of those modules (and the dependent systems) which do not conform to the new constraint. These actions are performed in the procedure `Initialize_Status_From_Dependencies`, shown in Figure 19.

In the first part of the procedure the tuples for `Actual_Dependency` are entered by a procedure that analyzes existing source code modules. Once this relation is initialized, the status of previously `accepted` modules is checked. If any do not satisfy the new dependency criteria they are rejected, as are the systems that depend on them.

This procedure also illustrates the use of the `enforced` attribute of APPL/A predicates and the related predefined operations. When the new mandatory predicate `Accepted_Code_Dependency_Conforms` is introduced it is automatically enforced by default. It is not initially violated, however, because the relation `Actual_Dependency` is empty and the condition tested by the predicate is trivially true. As `Actual_Dependency` is initialized, though, the predicate may be violated. Such a violation would prevent further updates to any of the relations referenced in the predicates, including `Actual_Dependency` (unless those updates happened to leave the predicate satisfied). To counter this possibility, an initialization procedure can turn off the default enforcement of the predicate. It can do this by acquiring a capability for the predicate, then using that capability in the predefined predicate operation `enforced` to set the corresponding predicate attribute `enforced` to false (thus turning off the enforcement). So long as the initialization procedure holds the capability no other process can acquire it to turn the enforcement of the predicate back on.

The initialization of `Actual_Dependency` and the repair of predicate violations are then performed while the predicate is not enforced. During this process, serial statements are used to read and write the relevant relations. This precludes interference by other processes, for example, competing changes to module status. At the conclusion of the initialization procedure, once consistency with the predicate has been established, the enforcement of the predicate is turned back on and the capability is released. An advantage of this approach, as opposed to a suspend statement, is that a failure in the initialization process will not result in rollback with

```
procedure Set_Status_From_Dependencies is
-- Analyze intermodule dependencies for
-- accepted modules in Source_Repository
-- and set code status accordingly.
  capability: integer;
  procedure Analyze_And_Record_Dependencies(
    module: name_type) is separate;
Begin
  -- turn off predicate which constrains code
  -- status wrt intermodule dependencies
  capability :=
    Accepted_Code_Dependency_Conforms'acquire;
  Accepted_Code_Dependency_Conforms'enforced(
    capability, false);

  serial read Allowed_Dependency,
    Actual_Dependency;
  begin
    -- Initialize Actual_Dependency
    serial write Actual_Dependency;
    begin
      for sr_t in Source_Repository loop
        analyze_and_record_dependencies(
          sr_t.name);
      end loop;
    end serial;

    -- Check dependencies, set status
    serial write Module_Status;
    begin
      for fixed ms_t in Module_Status where
        ms_t.code_status = accepted
      loop
        if not dependencies_consistent(
          ms_t.name) then
          ... -- code_status => rejected
              -- compile_status => outdated
        end if;
      end loop;
    end serial;
  end serial;

  -- turn on predicate; release capability
  Accepted_Code_Dependency_Conforms'enforced(
    capability, true);
  Accepted_Code_Dependency_Conforms'release(
    capability);
End Initialize_Status_From_Dependencies;
```

Figure 19: Procedure `Initialize_Status_From_Dependencies`

the consequent loss of work.

This example addresses just a few more aspects of change management in software processes and process programs. The scenario demonstrates the reuse of data (e.g., modules and systems). It does not preclude the conservation of off-line manual activities (e.g., programmers who were working on modules when the initial process was killed may submit those modules to the revised process). The principal difference between the old and new programs is the addition of the constraint on accepted modules, i.e., a change from one constraint system to another. The procedure presented in this section implements a systematic transition between those systems.

5 Related Work

The use of an example called "code and compile" invites comparison with Make [15]. Make represents the seed of an idea about process management whereas process programming represents the flowering of that idea. Make offers certain basic but comparatively limited capabilities; APPL/A, as a process-programming language, offers more general capabilities. Make is oriented toward automated tools and the maintenance of derivation dependencies, it has a restricted language for representing these dependencies, it performs only limited inferencing, and it manages objects using the host file system. These capabilities can be stretched to represent a surprisingly wide range of activities and objects, but in general they are inadequate for process programming. In contrast, APPL/A is based on a full conventional programming language, it includes specialized extensions for process programming, it provides a more abstract model of persistent data, and it includes triggers and a sophisticated model of constraints and transactions. It is intended to represent and support a wide range of relationships, and to coordinate both manual and automated activities that may involve multi-step inferencing and concurrency by multiple tools and developers.

The PMDB+ prototype project [28] was an exercise to extend the TRW Project Master Data Base [29] to include elements of process modeling and enaction. The PMDB project attempted to model the life-cycle process by representing its objects and relationships; it did not include activities. The PMDB+ model extends the basic PMDB model to address activities using operations that are associated with entity types. The PMDB+ prototype makes use of VBase [3], an object-oriented system, to support its entity-relationship model. VBase supports notions of relationships and triggers, but these are very different from their counterparts in APPL/A. VBase also has only a rudimentary transaction model and does not support constraints. A detailed comparison of VBase and APPL/A is found in [35].

A number of other systems have been developed recently to support software development, including the representation and enactment of software processes. These include (among others) Melmac [14], OS/O [43], and Oikos [2]. Each of these makes a significant contribution to change management in software processes by providing some formal representation of the software process and supporting the automatic enactment of that representation. They differ with respect to modeling formalisms and to specific features designed to support change.

Melmac provides a three-level approach to process representation. The top level is a C-oriented object and activity type definition, the middle level is an extended, graphical Petri-net model, known as FUNSOFT nets, which incorporates the objects and activities defined at the top level, and a bottom level which supports enactment of the FUNSOFT nets to execute or simulate the process. APPL/A does not include any support for graphical representations of processes, although process visualization research is an ongoing part of the Arcadia process. The FUNSOFT net model includes predicates which constrain the firing of agencies in the net, but, unlike APPL/A predicates, they do not apply directly and independently to constrain stored objects. FUNSOFT nets also allow the emulation of post-activity triggers by arcs which lead into further activities. Although Melmac is built on top of a persistent object store, the FUNSOFT net model does not seem to offer special support for the modeling of transactions in the software process. The FUNSOFT net model provides special support for change management in the form of "modification points." These are special nodes in the FUNSOFT net which are not completely specified at process-invocation time. When encountered during process execution they are elaborated dynamically, allowing the process to adapt itself to prevailing circumstances. Support for unelaborated activities in APPL/A programs can be modeled by stubs which call out to a user (or other process) which can then execute a separate program to "fill in" the unelaborated activity. APPL/A provides a related kind of flexibility through the ability to dynamically allocate tasks, triggers, and relations.

OS/O is a prototype object-management system based on the concept of object-oriented attribute grammars (OOAGs), which provide several extensions of conventional attribute grammars. In the OOAG model, trees are viewed as distinguished objects which respond to messages. Each object definition may have both a static and dynamic part which govern the computation of its attributes. Attributes may have persistent but modifiable values over multiple computations of the tree. The OOAG model has several features that are specifically designed to support management of the change of object class definitions. An object

class may have several alternative definitions, any one of which may be in effect. An object in such a class may be sent a "transform" message to change the effective definition from one to another alternative. Additionally, a class may be sent a "change" message to add or delete alternative definitions or to modify an existing definition. OS/O also supports persistent meta-objects which maintain version histories for versionable objects. APPL/A does not have any automatic versioning system. We have regarded versioning as a process-dependent activity, and so we have tried to provide constructs which allow support for versioning to be programmed. APPL/A can support an effect like that of alternative class definitions for objects in that relations can be used to associate object identifiers to alternative sets of attributes which are derived in different ways. The consistency of these different views of an object (as well as other kinds of consistency) can be maintained by enforced predicates, a mechanism which is not available in OS/O. Additionally, OS/O, like Melmac, does not provide special support for the modeling of transactions in software processes.

Oikos is an infrastructure for experimentation and evaluation of process models. The coordination language of Oikos is ESP, an extended distributed logic language based on Prolog. An ESP program defines a hierarchy of blackboards. A blackboard may contain facts and other blackboards, and it may have attached agents. An agent responds to facts on the blackboard to which it is attached; an agent can also create additional blackboards. Oikos distinguishes between processes and environments. In a process, the structure of subprocesses and use of tools is foreseen and frozen, whereas an environment offers a set of tools and services which may be used in an unstructured way. Where change is anticipated in a process, the process may embed an environment. The APPL/A model of process and program is based on very different conceptual elements. However, APPL/A does include active and reactive constructs, and it does allow the programming of processes which are comparatively constrained or unconstrained in their pattern of control.

Sullivan and Notkin [38] propose an approach to environment integration which supports flexibility through component independence. Their approach is based on mediators, which localize relationships, and a general event mechanism. The typical role of a mediator is to maintain a relationship between separate components; the mediator can respond to events that signal changes in those components by taking actions that maintain the intended relationship. This approach separates components from the relationships in which they participate and allows independent access to the components. New relationships can be added without affecting existing components or other mediators. Triggers in APPL/A are analogous to mediators, and APPL/A admits a similar approach to system integration and evolution. The event mechanism proposed by Sullivan and Notkin is more general than that in APPL/A. APPL/A provides an additional mechanism, however, in the form of predicates. These can be used to state explicit conditions that must hold between relations and that might be maintained by triggers (or some other mechanism). The system of Sullivan and Notkin does not include any way to state the conditions that relations are to satisfy or that mediators are to enforce.

AP5 [12, 13] has many features in common with APPL/A. It is an extension of common lisp that includes relations and rule-based constraints and triggers. These features enable AP5 to be used to model software processes, including managing change, in much the same way as APPL/A. However, AP5 is primarily a single-user system. It relies on virtual memory for data storage, and it provides only a simple transaction mechanism. AP5 also uses state-driven triggers, whereas APPL/A uses event-driven triggers.

Marvel [21, 20] is a rule-based kernel for software development environments. It provides many capabilities that support change management. Software processes are modeled by rules that encapsulate development activities and assist human users. The kinds of assistance available range from automation to consistency maintenance. The data model comprises a hierarchy of object classes with multiple inheritance. Objects may have attributes linking them to other objects (thus representing relationships between objects). A special goal of the project is to support a variety of modes of interaction between developers on large-scale projects.

6 Status and Future Work

APPL/A is defined as an extension to Ada [39]. The APPL/A definition includes a formal syntax and English semantics with examples in a style similar to that of the Ada manual [45]. An automatic APPL/A to Ada translator, called "APT", exists in prototype form. It is a modification of a partial Ada compiler which is a component of existing Arcadia [44] Ada language technology. For historical and technical reasons the language translated by APT is a subset of the full APPL/A language. APT can automatically translate relation specifications, relation bodies, triggers (with global event signaling) and predicates. APT can recognize the transaction-like statements, but it cannot translate them because of lack of run-time support capabilities. Support is also lacking for predicate enforcement and non-global event signaling (a feature not described above). However, we have developed designs for the implementation of all of these features, and the automatic translation of the full language is feasible. A practi-

cal problem at this time is that most of the features which are not yet translated depend on the ability to trace the execution of a (concurrent) program through its call stack. This information is difficult to obtain from current Ada compilers (without being able to modify them), and the development of an alternative approach, while possible, is costly. Future work will continue the evolution and implementation of APPL/A.

We recognize that for many users it is a burden to construct the bodies of relations. In light of that, we are designing default implementations for APPL/A relations based on some existing database systems. These database systems provide persistence and some form of data model into which APPL/A relations are mapped. Our original process programs used Cactis [19]. Current work is making use of Triton [17], a persistent object system built on top of the EXODUS [7] storage manager. It is now possible to generate default bodies, which use Triton, for simple stored relations (i.e. relations lacking derived attributes). We also plan to support default implementations for relations with derived attributes.

A goal for the APPL/A project is to code process programs covering a complete software life cycle. At least partial code exists to support requirements (the REBUS program) and design (DEBUS). REBUS is an executable system which supports the specification of software requirements in a functional hierarchy. REBUS stores data about requirements in APPL/A relations which are constrained by APPL/A predicates and maintained by APPL/A triggers. REBUS is translated by APT and makes use of Triton. DEBUS, a design-process support system, is under construction in APPL/A. DEBUS is based on the Yourdon Design methodology [47]. An early version of APPL/A was also used to code DataFlow⋈Relay, which integrates dataflow and fault-based testing and analysis [30]. REBUS and DEBUS were also initially coded using early versions of APPL/A. In the case of REBUS, the APPL/A code was translated by hand into an executable Ada program. The early version of REBUS also included extensions based on RSL/REVS [1, 5] which will be incorporated into the current version. The experience gained with the earlier process programs contributed greatly to our understanding of PPL requirements. That experience enabled us to refine APPL/A in significant ways, resulting in the definition described here. We have also used APPL/A to program a solution to the process-modeling problem presented in conjunction with the Sixth International Software Process Workshop (ISPW6). Ongoing and future work will include the development of programs for additional parts of the software life-cycle, such as coding and project management.

Change is pervasive in the programs mentioned above, and many of the programs have a significant component which explicitly addresses change management in some form. For example, REBUS supports and coordinates the development and modification of a requirements specification by a team of developers, and the ISPW6 problem treats changes to unit design, code, and test plans following a change to requirements. Consequently, the coding of these programs in APPL/A has allowed us to experiment with and evolve the constructs in the language and to verify that they do generally support the management of change in software-process programs. The coding of these processes has also brought to light several additional issues related to change management, for example, the capturing and use of information about change to processes and products (process reification and reflection), the design of process programs for ease of change to processes, and issues of language, environment, and process support for transitions between processes. As we develop additional process programs we plan to address these issues in greater depth, and we hope and expect to uncover further issues related to management of change in software development.

7 Acknowledgements

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. The authors wish to thank Deborah Baker, Roger King, Shehab Gamalel-din, Mark Maybee, and Xiping Song for their advice. The comments of the members of the Arcadia consortium were also important in clarifying the issues surrounding APPL/A.

References

[1] Mack W. Alford. A requirements engineering methodology for real-time processing requirements. *IEEE Trans. on Software Engineering*, SE-3(1):60 – 69, January 1977.

[2] 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.

[3] Timothy Andrews and Craig Harris. Combining language and database advances in an object-oriented development environment. In *OOPSLA '87 Conf. Proc.*, 1987.

[4] 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.

[5] T. E. Bell, D. C. Bixler, and M. E. Dyer. An extendable approach to computer-aided software requirements engineering. *IEEE Trans. on Software Engineering*, SE-3(1):49 – 59, January 1977.

[6] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, M. Muralikrishna, Joel E. Richardson, and Eugene J. Shekita. The architecture of the exodus extensible dbms. In *Proc. of the International Workshop on Object Oriented Database Systems*, pages 52 – 65, 1986.

[7] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, Joel E. Richardson, Eugene J. Shekita, and M Muralikrishna. The architecture of the EXODUS extensible DBMS: a preliminary report. Technical Report Computer Sciences Technical Report #644, University of Wisconsin, Madison, Computer Sciences Department, May 1986.

[8] Thomas E. Cheatham, Jr. The E-L software development database – an experiment in extensibility. In *Proc. 1989 ACM SIGMOD Workshop on Software CAD Databases*, pages 21 – 25, 1989. Napa, California, February.

[9] Thomas E. Cheatham, Jr. Process programming and process models. In *5th International Software Process Workshop – Preprints*, October 1989. Kennebunkport, Maine, October, 1989.

[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] 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.

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

[16] Shehab A. Gamalel-din and Leon J. Osterweil. New perspectives on software maintenance processes. In *Proceedings of the Conference on Software Maintenance*, pages 14 – 22. IEEE, October 1988.

[17] Dennis Heimbigner. Triton reference manual. Technical Report CU-CS-483-90, University of Colorado, Department of Computer Science, Boulder, Colorado 80309, August 1990.

[18] Dennis Heimbigner and Steven Krane. A graph transform model for configuration management environments. In *Proc. Third ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 216 – 225, November 1988. Special issue of SIGPLAN Notices, 24(2), February, 1989.

[19] Scott E. Hudson and Roger King. The Cactis project: Database support for software environments. *IEEE Trans. on Software Engineering*, 14(6):709–719, June 1988.

[20] 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.

[21] 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.

[22] Lech Krzanik. Enactable models for quantitative evolutionary software processes. In Colin Tully, editor, *Proc. 4th International Software Process Workshop*, pages 103 – 110, Moretonhampstead, Devon, U.K., May, 1988. ACM SIGSOFT Software Engineering Notes, v. 14, n. 4, June 1989.

[23] Yoshihiro Matsumoto, Kiyoshi Agusa, and Tsuneo Ajisaka. A software process model based on unit workload network. In *5th International Software Process Workshop – Preprints*, October 1989. Kennebunkport, Maine, October, 1989.

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

[25] K. M. Olender. *Cecil/Cesar: Specification and Static Evaluation of Sequencing Constraints.* PhD thesis, University of Colorado, 1988.

[26] Leon J. Osterweil. Using data flow tools in software engineering. In Muchnick and Jones, editors, *Progarm Flow Analysis: Theory and Application.* Prentice-Hall, Englewood Cliffs, N. J., 1981.

[27] Leon J. Osterweil. Software processes are software too. In *Proc. Ninth International Conference on Software Engineering*, 1987.

[28] Maria H. Pendeo. Acquiring experiences with executable process models. In *5th International Software Process Workshop – Preprints*, October 1989. Kennebunkport, Maine, October, 1989.

[29] Maria H. Penedo. Prototyping a project master database for software engineering environments. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1–11. ACM, 1986.

[30] 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.

[31] Joel E. Richardson and Michael J. Carey. Programming constructs for database system implementation in EXODUS. In *Proc. ACM SIGMOD Conf.*, pages 208–219, 1987.

[32] Mark J. Rochkind. The source code control system. *IEEE Trans. on Software Engineering*, SE-1:364 – 370, December 1975.

[33] Lawrence A. Rowe and Michael R. Stonebraker. The POSTGRES data model. In *Proc. of the Thirteenth International Conf. on Very Large Data Bases*, pages 83 – 96, 1987.

[34] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *OOPSLA '86 Conf. Proc.*, pages 9–16, 1986. Available as ACM SIGPLAN Notices 21, 11, November 1986.

[35] Christine Shu. Experience with using VBase and APPL/A for process modeling and programming. Arcadia Document Arcadia-TRW-89-021, TRW Corp., Redondo Beach, California, January 1990.

[36] John M. Smith, Steve Fox, and Terry Landers. Reference manual for ADAPLEX. Technical Report CCA-83-08, Computer Corporation of America, May 1981.

[37] Michael Stonebraker and Lawrence A. Rowe. The design of POSTGRES. In *Proc. of the ACM SIGMOD International Conf. on the Management of Data*, pages 340 – 355, 1986.

[38] 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.

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

[40] 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.

[41] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. APPL/A: A prototype language for software process programming. Technical Report CU-CS-448-89, University of Colorado, Department of Computer Science, Boulder, Colorado 80309, October 1989.

[42] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. Language constructs for managing change in process-centered environments. In *Proc. of the Fourth ACM SIGSOFT Symposium on Practical Software Development Environments*, pages 206–217, 1990. Irvine, California.

[43] Lichao Tan, Yoichi Shinoda, and Takuya Katayama. Coping with changes in an object management system based on attribute grammars. In *Proc. of the Fourth ACM SIGSOFT Symposium on Practical Software Development Environments*, pages 56–65, 1990. Irvine, California.

[44] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon J. Osterweil, Richard W. Selby, J. 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.

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

[46] Alexander L. Wolf, Lori A. Clarke, and Jack C. Wileden. The adapic toolset: Supporting interface control and analysis throughout the software development process. *IEEE Trans. on Software Engineering*, 15(3):250–263, March 1989.

[47] Edward Yourdon. *Techniques of Program Structure and Design.* Prentice Hall, 1975.