FCM:
A Flexible Consistency Model for Software Processes *

Stanley M. Sutton, Jr.

CU-CS-462-90

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* 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.

FCM:
A Flexible Consistency Model for Software Processes

Stanley M. Sutton, Jr.
Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430

March, 1990

CU-CS-462-90

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.

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR AND DO NOT NECESSARILY REFECT THE VIEWS OF THE NATIONAL SCIENCE FOUNDATION

Contents

1 Introduction **5**

2 Consistency in Software Processes **6**

2.1 Arguments for Flexibility in Determining *When* Predicates are Enforced 7

2.2 Arguments for Flexibility in Determining *Where* Predicates are Enforced 9

2.3 Arguments for Functional Flexibility 10

2.4 Flexibility to Accommodate Inconsistency 12

2.5 Summary of Arguments 12

3 Requirements for a Flexible Consistency Model **13**

4 A Flexible Consistency Model: FCM **14**

4.1 Execution Model 14

4.2 Predicates and Consistency 15

4.3 Control Constructs 23

4.3.1 Overview 23

4.3.2 The Consistency Management Statements 25

4.4 Composition of the Statements 28

4.4.1 Combined Composed Constructs 29

4.4.2 Separate Composed Constructs 30

5 Examples **31**

5.1 Some Combinations of Consistency Management Statements 33

5.2 Managerial Control of Consistency Maintenance 34

5.3 Federated Control of Consistency Maintenance 38

5.4 Cooperative Work 44

6 Discussion **48**

6.1 Comparison of FCM and the Requirements 51

6.2 Inconsistency 51

6.3 Serializability and Deadlock 52

7 Related Work **55**

7.1 Balzer's "Accommodating Inconsistency" 55

7.2 Moss's Nested Transactions 56

7.3 Abstraction 56

7.4 Constraint Enforcement by Triggers 57

7.5 Pu, Kaiser, and Hutchinson's Split- and Join-Transactions 57
7.6 Korth and Speegle's Correctness without Serializability 59

8 **Summary** **60**

9 **Acknowledgements** **62**

List of Figures

1 Predicates in an APPL/A-like Syntax 17
2 Predicate and Consistency Example – Procedure 1 18
3 Predicate and Consistency Example – Procedure 2 19
4 Sketch of Procedure Test_and_Analysis_Suite 32
5 Testing Procedure with a Separate Included Transaction 33
6 Sketch of a "Conventional Flat Transaction" in FCM 34
7 Sketch of a "Nested Transaction" in FCM 35
8 Sketch of an "Assertion" Transaction in FCM 36
9 Sketch of a "Repair-Enforce Statement" 36
10 Sketch of Relations and Predicates for Requirements Process 39
11 Sketch of Subprocedures for Requirements Process 40
12 Sketch of Requirements_Manager Procedure – Part 1 41
13 Sketch of Requirements_Manager Procedure – Part 2 42
14 Sketch of Relations, Predicates, and Procedures for Federated Software Processes 43
15 Sketch of Typical Procedure Body for Federated Software Processes – Part 1 45
16 Sketch of Typical Procedure Body for Federated Software Processes – Part 2 46
17 Sketch of Procedure Cooperative_Coding 47
18 Sketch of Programmer Task Body – Part 1 49
19 Sketch of Programmer Task Body – Part 2 50

List of Tables

1 Summary of Capabilities for FCM Consistency-Management Statements . . . 24

1 Introduction

This paper presents a flexible model of consistency for software processes and products. The model is motivated by the difficulty of defining and maintaining the consistency of software products during software development. Software development can be viewed as the process of creating a consistent software product. However, software processes are lengthy and complex, the criteria for consistency are often dynamic and relative to specific processes, and inconsistency is often inescapable. (A detailed example is presented in Section 2.) The goal of the flexible consistency model presented here is not to attempt to suppress these problems. Rather it is to accommodate the problems of representing and maintaining consistency in a way that facilitates the modeling of software processes and the development of software products.

A consistency model for software products has several aspects. It must minimally include some notion of consistency for those products and some mechanism for evaluating that consistency. In practice the criteria for consistency may be implicit or explicit, and the mechanism for evaluating and enforcing consistency may be manual or automatic (for example, see [8,19,11,12,10]). The model may also include some view, either implicit or explicit, of how the criteria for consistency evolve in time (if at all). A practical consistency model must also be integrated with a model for operations on the data, and it must include rules about the consequences of consistency (or inconsistency) for those operations. Operations on data are typically addressed in "transaction models", which may also include operational criteria for consistency i.e., serializability and atomicity [13,14,18,12]. In the conception of this paper a general consistency model subsumes a transaction model. The capabilities in each area complement one another, and flexibility in both is regarded as essential for software processes.

This paper is organized as follows.

- Section 2 discusses consistency in software development and presents arguments in favor of a flexible consistency model for software processes. These arguments are motivated by examples from a hypothetical software development scenario. These examples include aspects of both the definition and management of consistency.

- Section 3 briefly sets out general goals and specific requirements for a flexible consistency model for software processes, including a flexible transaction model.

- Section 4 presents a flexible consistency model, "FCM", which addresses these requirements. FCM is presented from the perspective of software process programming [15], and the model is assumed to be embedded in a software-process programming language. The model includes both predicates for consistency representation and transaction-like constructs for data manipulation.

- Section 5 presents examples of the use of features in FCM.

- Section 6 discusses several issues in the design and use of the model. These include comparison of the model with the requirements, inconsistency, serializability, and deadlock.

- Section 7 discusses related work and compares it with FCM.

- Section 8 summarizes the paper.

This model has been developed as part of the ongoing design of APPL/A [23], a prototype software-process programming language based on Ada [1]. APPL/A extends Ada with persistent relations with programmable implementations and derived attributes, triggers on relation operations, predicates over relations, and several transaction-like control constructs for composite operations on relations. The APPL/A predicates and control constructs are based on FCM, and APPL/A contains specific approaches to many of the general concepts introduced here; APPL/A also addresses recovery management more fully. However, the details of APPL/A are beyond the scope of this report. An APPL/A-like syntax is used to present many of the FCM examples. These examples are intended to illustrate FCM, not APPL/A per se, so the syntax is simplified for this purpose.

2 Consistency in Software Processes

The consistency of software products is an evolving and relative condition in software processes. Moreover, the inconsistency of software products is common and natural occurrence during software development. This section presents arguments that a flexible model of consistency is necessary to enable software processes to accommodate these situations.

Conventional databases with global constraints and transactions provide a basic model of consistency. In this model the constraints on the database are predetermined. The constraints may be explicit or implicit, but all processes that operate on the database must observe them. Transactions are the mechanism by which the database is updated. When a process executes a transaction it is guaranteed serializable access to the data, i.e. it executes as if no other processes were executing concurrently. During the transaction, constraints on the database are temporarily suspended and may be violated. Upon completion of the transaction, constraints must be again satisfied; if they are not, the results of the transaction are undone, or rolled back. A transaction may also be rolled back as a consequence of other errors, e.g. the raising of an exception during a computation. Because transactions are serializable, no process outside the transaction can see the possibly inconsistent state that holds within the transaction. Transactions always begin and end with the database in a consistent state, and no transaction sees any inconsistency which it does not itself create.

The conventional database model of consistency and transactions can be considered with respect to software processes. This model is widely recognized as inadequate for software object-management [4,5,16,18]. However, it does provide a minimum degree of flexibility in modeling the consistency of software products and some aspects of software object management. Additionally, the conventional model can be generalized and extended to help derive a flexible model of consistency that is more appropriate to software processes.

Arguments for a flexible consistency model for software processes are best developed with respect to examples of software development. The following paragraphs describe hypothetical software processes and products that provide a basis for the subsequent discussion. These processes and products are not intended to be all-encompassing; rather, they provide particular illustrations of the kinds of situations which may arise during software development in general.

Consider a software environment in which various software processes are executing concurrently. These processes include both automated and manual components. More specifically, a requirements process is composing the requirements for a given software product and a design process is constructing the design for that product. Both of these processes are composed of several subprocesses which perform particular tasks.

The requirements are represented as graphs of requirements elements. Each element corresponds to requirements in one or more areas such as functionality, performance, security, and so on. The graphs represent various relationships among the elements, including functional decomposition, "uses", and other relationships. The design is similarly represented by graphs of design elements.

Each kind of element and graph is characterized by certain predicates. For example, each requirements element must have a unique, non-null name, it must have an author who is assigned to the project, and it must have a creation date later than the start date of the project. The requirements graphs may be required to be connected trees or DAGS. Comparable predicates characterize the design elements and graphs. The *consistency* of the requirements and design is judged with respect to such predicates. Some predicates constitute acceptance criteria for the final product. These comprise overall goals for the requirements and design processes. Other predicates serve as intermediate goals or as preconditions for the correct execution of development processes.

The following subsections present arguments for flexibility in determining when predicates are enforced, flexibility in determining where predicates are enforced, functional flexibility for managing objects and maintaining consistency, and flexibility in accommodating inconsistency.

2.1 Arguments for Flexibility in Determining *When* Predicates are Enforced

When predicates on software objects are enforced they can be regarded as constraints. However, it is impractical to strictly enforce all predicates at all times. For example, consider the (potential) constraint that a requirements element have a unique, non-null name. This condition is relevant from the beginning of development, and virtually all processes that use requirements data will depend on it. However, even this simple and basic condition will be violated when a requirements element is being created or edited (e.g. before a name is assigned, or while a name is being changed). Thus a minimal degree of flexibility is needed to suspend constraint enforcement during restricted periods so that composite updates can be treated atomically for purposes of consistency maintenance.

In conventional databases this flexibility is provided by transactions in which constraint enforcement is temporarily (and locally) suspended. A transaction-like mechanism is appropriate in the case described above because the condition to be enforced is widely important and because periods of inconsistency are usually brief and well delimited. Moreover, this situation is well modeled by the conventional database approach because the constraints are known in advance and are enforced except during brief periods. Thus the conventional database model does have applicability in software processes for short-term control of consistency. However, a notable limitation of the transaction-based model is its lack of flexibility with respect to the longer-term evolution of consistency.

The criteria for consistency of software products may evolve over time because not all potentially relevant constraints are known in advance, and not all known constraints are relevant or usefully enforced during all phases of development. For example, constraints on the overall structure of requirements and design graphs may only be relevant once the graphs are nearly complete. Early on in the requirements process it may be desirable to identify requirements elements without worrying about the overall requirements structure; a constraint on the graph structure during this phase may be irrelevant to most processes active at that time and it may actually impede the task of identifying requirements. Thus, even though the constraints are known at the outset of development, they may only be relevant during the later phases.

Consider also the case of a constraint that is identified during the course of development. For example, suppose that the design of the product turns out to be unusually complex. To simplify the design a limit may be imposed on the branching factor of the design graphs. This limit comprises an entirely new criterion for consistency of the software product, one which could not have been enforced previously.

For these situations the conventional transaction-based view of consistency is inadequate. The processes during which the enforcement of constraints is to be suspended are lengthy and open-ended, and there may be little need to enforce the constraints to begin with. Moreover, constraints may be added (and deleted) dynamically during the course of development.

A more natural approach is to consider that some constraints (like those on the connectedness of graphs) simply don't apply during some phases of development. What seems more appropriate are constraints that can be "turned on" when they are applicable and "turned off" when not. This view of constraints accommodates the case in which an entirely new constraint is added during development or an existing constraint is dropped completely. The use of optionally-enforcible constraints would allow the criteria for consistency of software products to evolve over time in a flexible way. It would enable software processes to be conceived and expressed more naturally, and it would free processes from the burdens of managing constraints that are irrelevant.

2.2 Arguments for Flexibility in Determining *Where* Predicates are Enforced

The above discussion suggests that software processes would benefit from consistency mechanisms that enable flexible control over *when* predicates are enforced. Further considerations suggest that software processes would also benefit from consistency mechanisms that enable flexible control over *where* predicates are enforced, i.e. flexible control over the scope of predicate enforcement.

In the conventional database model, constraints apply over the whole database and hence equally to all processes that operate on the database. However, each transaction on the database *locally* suspends the enforcement of constraints so that composite updates that temporarily violate constraints can be made atomically. Global consistency is maintained because each transaction is *serializable*, i.e. it executes as if in sequence with other transactions. Serializability precludes the possibility that one concurrent transaction can see an inconsistent state that may obtain during another transaction.

This model of global, process-independent constraints which are temporarily and locally violated is also applicable in software processes. Consider again the constraint that requirements elements have non-null, unique names. A requirements subprocess that creates and updates requirements elements must locally suspend this constraint. A design process that simply makes use of the requirements elements may depend on the satisfaction of this predicate for correct operation. The predicate violations induced by the requirements subprocess can be "hidden" within a serializable, transaction-like construct, and the design process (and others) can access the requirements elements with the assurance that constraints on element names are satisfied.

While the conventional, transaction-based model of consistency is applicable in software processes, it is not all together sufficient with respect to control over where predicates are enforced. It enables all constraints to be suspended locally, but it does not address the complementary case in which an unenforced predicate must be treated like a constraint. Such predicates may be preconditions to the correct execution of a process, and they may be treated like assertions within the process. For example, a design process may identify design elements based on analysis of a requirements graph, and that analysis may require that the graph be connected. This condition should be assured when the design process begins its analysis, and if found to be violated during the analysis then the design process should be abort.

There are at least two general reasons why a predicate which serves as a precondition for a process may not be enforced as a global constraint. First, it may not be enforced at a given time because it is not relevant to other processes that are executing. In this case, despite the fact that the predicate is not generally enforced, the process which requires the predicate to be satisfied may still execute opportunistically if conditions permit. Second, the predicate may not be regarded as a global constraint at any time; it may be relevant only as the precondition of a particular process. To cope with these circumstances some mechanism should be available that allows a process to *locally* enforce a predicate that is not enforced globally. Such a mechanism would to some extent be the converse of conventional transactions, which locally suspend enforcement. In combination these mechanisms would enable a software process to locally suspend or enforce predicates as required for its particular purpose, thus generalizing the flexibility provided by conventional transactions.

2.3 Arguments for Functional Flexibility

Conventional transactions provide a combination of capabilities:

- Concurrency control, i.e. serializable access to data (typically differentiating read and write access).

- Temporary suspension of constraint enforcement.

- Atomicity, i.e. all-or-nothing execution of possibly composite operations (with rollback in the event of incomplete or inconsistent results).

Thus conventional transactions can be regarded as units of concurrency control, units of consistency, and units of completeness of work.

A criticism of conventional transactions with respect to software object management is that they provide all these kinds of functionality where more specialized capabilities may be sufficient. In [17] it is suggested that separate units of concurrency control and consistency should perhaps be provided for software object management. Taking this suggestion further, it may be hypothesized that software processes require a variety of kinds of "units" for object management and consistency maintenance: these units would provide relatively specialized functionality, but should be composable to achieve the effects of conventional transactions or other combinations of capabilities (e.g. nested transactions [13]).

The following examples illustrate the use of capabilities that are relatively specialized compared to conventional transactions.

- A requirements subprocess performs connectivity analysis of the requirements graphs. It requires serializable read access to those graphs. However, it does not update the graphs and so its own operations do not need to be atomic. Additionally, since its purpose is to assess the state of the requirements "as-is", it requires no special context with respect to predicate enforcement.

- A design subprocess opportunistically analyzes requirements graphs to obtain information for a preliminary design. This process begins to operate while work on the requirements graphs is nearly but perhaps not entirely complete. This process depends on the graphs being acyclic, so it requires a predicate to this effect to be enforced on itself (regardless of whether it is enforced globally). Because the process does not write the requirements graphs it does not need atomicity. However, because the design process is opportunistic and preliminary, it has a lower priority than any requirements processes that may still be operating. Consequently, as a matter of policy, the design process cannot obtain serial access to the requirement graphs (which would exclude requirements processes).

- A requirements subprocess performs semantic analysis of requirements graphs in preparation for a preliminary requirements review. This process obtains serial read access to the graphs so that it can examine them in a static state. The analysis depends on the requirements elements satisfying certain criteria of completeness, which may or may not be met at this early phase of development. Consequently, the process asserts for itself that the elements are "complete" and aborts if it discovers that one is not. Because the process does not write the graphs or individual elements, it does not require atomicity.

- A design subprocess is reorganizing a preliminary design graph to remove cycles in anticipation of a future constraint against cycles. Because the process is writing the design graph, it requires serializable write access. Because there are no constraints on the graph structure at this time the process does not require any exceptional context with respect to predicate enforcement. However, because the process may introduce new cycles temporarily while removing existing cycles, it requires atomicity (i.e. the reorganization should not terminate in an incomplete state that may still contain cycles).

These examples illustrate the use of serializability alone, the use of specialized enforcement contexts alone, and the two combinations of serializability with specialized enforcement and with atomicity. Conventional transactions combine all three capabilities. This suggests that some mechanisms to obtain each of these individual and combined capabilities would be useful. Moreover, note that in the last example atomicity is required only to assure that the work is performed completely or not at all; it is not required to assure that existing constraints are satisfied. In conventional transactions atomicity is also used to enforce constraints. This illustrates that atomicity actually performs two roles which are differentiable: it serves to assure completeness of work and also to assure consistency of work. This further suggests that two kinds of "atomic" control constructs may be useful.

2.4 Flexibility to Accommodate Inconsistency

Inconsistency, for better or worse, is a common occurrence in software processes. One principal cause of inconsistency is that not all criteria for consistency are known in advance. The development activity entails the discovery of new constraints on final and intermediate products. When these constraints are imposed they may not be satisfied by existing software objects. Another cause of inconsistency is that software processes are relatively prone to error compared to conventional transactions (and computer applications in general). Many large-scale software processes, such as the specification of requirements of the development of design, are lengthy and open-ended. They include important manual subprocesses, and they may incorporate experimental procedures and systems. Moreover, they are subject to changes in environment, personnel, and project goals. All of these factors tend to promote results which are (at least temporarily) incorrect, inconsistent, and incomplete. Yet another cause of inconsistency in software processes are conflicts between processes. While a project overall may have a single, well-defined goal, the various subprocesses may have competing or conflicting subgoals. For example, a process to identify requirements may fulfill its task most effectively by ignoring the organization of those requirements, while other processes may impose or require a particular organization for them. A requirements process may organize requirements one way to promote their understandability, while a design process may attempt to reorganize them to facilitate their mapping to the design. Consequently, a software process may manage objects to make them consistent for one purpose while rendering them inconsistent for another. Inconsistency may arise in this way even if no new constraints are identified and all processes operate correctly.

The above considerations imply that inconsistency is inherent in software processes. If inconsistency is inescapable, then software processes may be well served by a consistency model in which inconsistency can be *accommodated.* The ability to accommodate inconsistency would give software processes a degree of flexibility in consistency management that is alien to conventional databases. As suggested by examples in the previous subsections, inconsistency may be accommodated by mechanisms to control when constraints are enforced or suspended globally for all processes, when predicates are asserted or denied locally for individual processes, when serializable access to data is required to preclude interference between concurrent processes, and when recoverable access to data is need to assure complete or consistent work.

2.5 Summary of Arguments

A rigid view of the consistency of software objects is inappropriate for software processes. In software processes the criteria for consistency evolve over time. They may also vary from process to process: different processes may have different preconditions and different goals with respect to the consistency of objects. Software processes may also require specialized functionality for managing objects and maintaining consistency in the face of evolving and possibly conflicting constraints. Moreover, inconsistency is natural to software processes and should be accommodated in a natural way. All of these considerations indicate that a flexible consistency model is required for software processes.

3 Requirements for a Flexible Consistency Model

The discussion above illustrates that several kinds of flexibility are needed in a model of consistency for software processes. These include:

- flexibility as to when predicates are enforced,
- flexibility as to where predicates are enforced,
- flexibility of functionality in managing objects, and
- flexibility to accommodate inconsistency.

These kinds of flexibility represent general goals which should be realized by any particular model of consistency for software processes.

Consideration of the details of the software processes in the preceding examples suggests the following, more specific requirements for a flexible consistency model to satisfy the above goals:

1. The ability to specify predicates on data that may be enforced both globally, i.e. for all processes, and locally, i.e. for individual processes.
2. The ability to "turn on and off" the default enforcement of both global and local predicates.
3. Specialized local control over of functionality:
   - serializable access
   - suspension of predicate enforcement, regardless of default enforcement
   - imposition of predicate enforcement, regardless of default enforcement
   - atomicity with respect to consistency and/or completeness of work

   Also the ability to compose these capabilities.

These requirement address each of the kinds of flexibility set forth as general goals. Flexibility as to when predicates are enforced is provided by the ability to turn predicate enforcement on and off and also by the ability to temporarily and locally suspend or impose predicate enforcement. Flexibility as to where predicates are enforced is provided by the ability to specify globally or locally enforced predicates and also by the ability to locally (and temporarily) suspend or impose predicate enforcement. Flexibility of functionality is provided by specialized control over serializability, predicate enforcement, and atomicity, plus the ability to combine these.

Flexibility in accommodating inconsistency is provided in several ways. The ability to turn off predicate enforcement globally or locally can allow processes to proceed in the face of inconsistency; this enables inconsistency to be ignored if not repaired. The ability to turn on predicate enforcement globally or locally can assure that processes proceed only when consistency obtains; this ensures that if the processes execute at all they will do so in a consistent state. The ability to effectively serialize access to data enables processes to avoid interference by other processes. Atomicity (the ability to rollback the results of operations) enables processes to undo results that may be be inconsistent or incomplete.

These requirements provide the basis for the flexible consistency model defined in the next section. That model, in turn, is realized concretely in the APPL/A programming language as described in the succeeding section.

4 A Flexible Consistency Model: FCM

This section presents a flexible consistency model for software processes. For convenience, this model is called "FCM." FCM is presented from the perspective of software process programming [15]. Software process programming is the representation of software development processes using formal programming languages. APPL/A [23] is an example of a prototype process-programming language. The model is defined in programming-language terms on the assumption that it will be realized in software-process (or other) programming languages. However, the general concepts should be adaptable to software environments and object-management systems.

This section begins with a subsection that describes the execution model to which the consistency model applies and in terms of which it is defined. This model is derived from programming languages. The consistency model itself is then presented in two parts: first a discussion of predicates and consistency, then the introduction of transaction-like control constructs for concurrency control, atomicity, and consistency management.

4.1 Execution Model

A transaction model serves to organize operations on data. Transactions group individual operations on data into a composite unit. The transaction model provides rules about how such operations are executed within a transaction and how transactions are executed in isolation and in combination. A transaction model thus defines (or is defined in terms of) an execution model.

Conventional and nested transaction models have been developed for database systems. These define execution in terms of primitive operations on data and transactions which compose primitive operations. FCM is a consistency model for programming languages. Consequently, it must be defined with respect to an execution model appropriate to programming languages, i.e. one which is defined in terms of programming-language constructs. Thus the conventional and nested transaction models and FCM all presume some execution model. The features of these execution models differ, however.

The conventional transaction model is concerned primarily with concurrency control among parallel processes which represent transactions. The transactions themselves are simple. They do not include nested subtransactions or subroutine calls. Operations on data are primitive reads and writes.

Nested transactions (in the widely used model of Moss [13]) extend the execution model of conventional transactions with nesting. Concurrency is allowed between top-level transactions and also between subtransactions within an enclosing transactions. The model does not include subroutines, and operations on data are still primitive reads and writes.

The execution model of FCM also includes concurrency and nesting as in the nested transaction model. However, the execution model of FCM incorporates two extensions that are commonly found in programming languages: subroutines and user-defined abstract data types. These features are not commonly addressed in transaction models. One exception is the work of Moss and others [14], which addresses recovery management for a hierarchical model of abstract operations. The POSTGRES data model also includes flat abstract data types (defined directly in terms of internal representations) in combination with flat transactions [20,22].

In the execution model proposed for FCM, subroutines can be called from transactions as a kind of dynamic subtransaction. User-defined abstract data types have non-primitive operations, that is, operations that are implemented in terms of operations on other types. Those other types may themselves be abstract, so that one abstract type may be implemented in terms of others. In FCM, the operations on an object, whether of a primitive or abstract type, are assumed to be atomic and serializable with respect to other operations on that object. The FCM consistency model is thus defined with respect to an execution model that includes concurrency, nesting, subroutines, and abstract data types.

4.2 Predicates and Consistency

This section describes the role of predicates in FCM and defines the FCM notion of consistency in terms of predicates and their enforcement.

Introduction to the Example The concepts in this section are illustrated using predicates and procedures presented in an APPL/A-like syntax. These predicates and procedures reference a relation Bin_Tree (not shown) which represents a binary tree. The relation Bin_Tree has the following attributes:

- name: the name of the node;
- data: data associated with the node;
- left_child and right_child: the names of the two children of the node.

Operations on the relation include insert, update, delete, and find, which, respectively, insert, update, delete, and retrieve tuples from the relation. The predicates characterize the state of the relation, while the procedures operate on that state. (Note that these procedures are only sketches for the sake of the example; no attempt has been made to give them clearly defined functionality.) The examples are discussed further below as relevant concepts are introduced.

Predicates In FCM the consistency of software objects is judged with respect to *predicates.* Predicates are distinguished class of boolean functions over (possibly selected classes of) software objects. In APPL/A, for example, predicates are predicate logic expressions over relations; the expressions can include both quantified and conditional constructs and calls to other boolean functions and predicates. Like other boolean expressions, predicates can be tested and used to control program execution.

Figure 1 includes three predicates, in an APPL/A-like syntax, that apply to the relation Bin_Tree. Predicate At_Most_One_Parent tests whether each child node in the tree has only one parent node. Predicate No_Dangling_Children tests whether there is a node for each named child. Predicate Data_OK applies an externally defined boolean function to evaluate the data at each node. Figures 2 and 3 show various uses of these predicates in procedures to update and analyze, respectively, the binary tree represented by relation Bin_Tree. Further details of the procedures and predicates are described below.

Predicate Enforcement A predicate over objects can be *enforced* with respect to a process. If a predicate is enforced with respect to a process then no operation by that process on the objects to which the predicate applies is allowed to terminate in violation of the predicate. Any such operation is undone and results in the raising of an exception. Thus an enforced predicate amounts to a postcondition for operations on objects. However, an enforced predicate need *not* be satisfied *during* an operation on an object to which the predicate applies. By analogy with conventional transactions, each operation on an object is regarded as a unit of consistency and atomicity with respect to predicates enforced on that object. During an operation on an object the predicates that apply to that object may be violated; however, other enforced predicates must still be satisfied. Upon completion of an operation on an object the enforced predicates that apply to that object must be satisfied

```
global enforced predicate At_Most_One_Parent is
    every t1 in Bin_Tree satisfies
        if t1.left_child /= ‘‘none’’ then
            no t2 in Bin_Tree satisfies
                ((t2.left_child = t1.left_child or
                 t2.right_child = t1.left_child) and
                 t2 /= t1)
            end no
        end if
    and
        -- similarly for t1.right_child
        ...
    end every;
end At_Most_One_Parent;


predicate No_Dangling_Children is
    every t1 in Bin_Tree satisfies
        if t1.left_child /= ‘‘none’’ then
            some t2 in Bin_Tree satisfies
                t1.left_child = t2.name
            end some
        end if
    and
        -- similarly for t1.right_child
        ...
    end every;
end No_Dangling_Children;


with data_check;        -- externally defined boolean function
Predicate Data_OK is
    every t in Bin_Tree satisfies
        data_check(t.data);
    end every;
end Data_OK;
```

Figure 1: Predicates in an APPL/A-like Syntax

```
with Bin_Tree, No_Dangling_Children;

procedure Proc_1 is
-- Update Bin_Tree structure
    i:  integer := No_Dangling_Children'acquire;
begin
    -- At_Most_One_Parent is ‘‘on’’ by default,
    -- No_Dangling_Children is ‘‘off’’ by default
    Bin_Tree.delete(...);          -- must satisfy At_Most_One_Parent only

    if not No_Dangling_Children then   -- test No_Dangling_Children
        Bin_Tree.insert(...);          -- must satisfy At_Most_One_Parent
        ...                            --   only
    end if;

    No_Dangling_Children'enforced(i, true);
                                   -- turn No_Dangling_Children ‘‘on’’
    Bin_Tree.update(...);          -- now must satisfy At_Most_One_Parent
    ...                            --   and No_Dangling_Children both

    suspend At_Most_One_Parent, No_Dangling_Children;
    begin
        -- update nodes and edges, temporarily violating predicates
        Bin_Tree.insert(...);      -- here needn't satisfy either
        ...
        enforce No_Dangling_Children
        begin
            -- adjust edges among existing nodes
            Bin_Tree.update(...);  -- here must satisfy No_Dangling_Children
            ...
        end enforce;
    end suspend;
    -- here must satisfy At_Most_One_Parent and No_Dangling_Children both
    ...
end Proc_1;
```

Figure 2: Predicate and Consistency Example – Procedure 1

```
with Bin_Tree, No_Dangling_Children, Data_OK;

procedure Proc_2 is
-- Analyze Bin_Tree data
    i:  integer := No_Dangling_Children'acquire;
begin
    No_Dangling_Children'enforced(i, true);   -- turn enforcement ‘‘on’’

    serial read Bin_Tree;      -- exclude writers during analysis
    begin
        -- repair tree structure prior to analysis
        if not (No_Dangling_Children and At_Most_One_Parent) then
            suspend No_Dangling_Children and At_Most_One_Parent;
                -- serial write access to Bin_Tree within suspend
                Bin_Tree.update(...);  -- needn't satisfy either predicate
                ...                    --   within suspend;
            end suspend;               -- must satisfy both upon completion
        end if;                        --   or rollback ensues

        -- perform the analysis
        enforce Data_OK;        -- must also satisfy Data_OK within enforce;
        begin                   -- if violated an exception will be raised
            -- analyze data     -- and the analysis will fail
            ...
        end enforce;
    end serial;
end Proc_2;
```

Figure 3: Predicate and Consistency Example – Procedure 2

or the operation is rolled back. In general the enforcement of predicates is optional and can be changed dynamically; mechanisms for controlling the enforcement of predicates are described below.

Predicate Extent A predicate has an *extent*, which is either global or not. A predicate can be declared *global.* The extent of a global predicate automatically subsumes all programs which use objects to which the predicate refers. A predicate that is not declared global is called *local.* A local predicate 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 deliberately included. In the example `At_Most_One_Parent` is global and is included implicitly in both `Proc_1` and `Proc_2` (both of which reference the relation `Bin_Tree`, to which `At_Most_One_Parent` applies). `No_Dangling_Children` and `Data_OK` are both local; `No_Dangling_Children` is include explicitly (but voluntarily) in both `Proc_1` and `Proc_2`, while `Data_OK` is included explicitly in `Proc_2` but omitted from `Proc_1`.

Default and Actual Enforcement; Enforcement Setting Contexts A predicate is potentially enforcible throughout its extent. Whether a predicate is actually enforced at a given point in the execution of a process depends on its *default enforcement* or the applicability of an *enforcement-setting* context.

- Each predicate has a default enforcement. The default enforcement is represented by a boolean attribute. If the value of this attribute is `true` then the default enforcement of the predicate is "on"; otherwise it is "off." The default enforcement of a local predicate is initially off at the start of each program execution. However, during program execution the default enforcement of a local predicate can be dynamically turned on when relevant (and turned off again when not). Mechanisms for this are described below. The default enforcement of a global predicate is initially on at the time of its creation. Unless declared **enforced** it can subsequently be turned on and off like a local predicate, but the default enforcement persists between program executions. Consequently, at the start of any program execution, the default enforcement of relevant global predicates may be either on or off depending on how it was most recently set.

- There are two kinds of enforcement-setting contexts. One is operations on objects. Within an operation on an object the predicates that apply to that object are not enforced. The other includes special block statements (described in Section 4.3). These mandate or suspend the enforcement of designated predicates, regardless of their default enforcement.

When no enforcement-setting context applies the actual enforcement of a predicate is determined by the default enforcement. Otherwise the actual enforcement is determined by the applicable enforcement-setting context for the predicate.

The applicability of an enforcement-setting context is determined both statically and dynamically:

- An enforcement-setting context for a predicate applies statically within the lexical scope of the construct that creates the context but excludes the lexical scopes of any nested constructs which also set the enforcement of that predicate. Note that this scope may include concurrent processes.

- An enforcement-setting context applies dynamically within any subprogram called directly or indirectly from within the lexical scope to which the context applies (but excluding the lexical scopes of any constructs within the subprogram which also set the enforcement for the predicate. However, applicability does not extend across entry calls into a concurrent processes with which a rendezvous may occur.

In the example the default enforcement of `No_Dangling_Children` and `Data_OK` is initially off in both procedures. The default enforcement of the global predicate `At_Most_One_Parent` is always on since the predicate is declared **enforced**. `Proc_1` includes two enforcement-setting contexts, a **`suspend`** statement with a nested **`enforce`** statement. The **`suspend`** statement locally suspends the enforcement of `At_Most_One_Parent` and `No_Dangling_Children`, while the **`enforce`** statement locally requires the enforcement of `No_Dangling_Children`. The example also shows the use of enforcement-setting statements. `Proc_2` includes a **`suspend`** statement for `No_Dangling_Children` and `At_Most_One_Parent` and an **`enforce`** statement for `Data_OK`. (`Proc_2` also includes a **`serial`** statement which does not affect predicate enforcement but which assures serializable read access to relation `Bin_Tree`.)

Extent of Default Enforcement The extent over which the default enforcement of a predicate applies depends on whether the predicate is global or local. A global predicate has a *collective* default enforcement which applies uniformly across all of the programs within the extent of the predicate. At any given time the predicate is enforced by default in either all or none of these programs. A local predicate has a non-collective or *individual* default enforcement for each program in the extent of the predicate. A local predicate is enforced (or not) in any one program independently of whether it is enforced (or not) in any other program.

In the example the global predicate `At_Most_One_Parent` is enforced collectively for both `Proc_1` and `Proc_2`. If this predicate were not declared **enforced** and the default enforcement were turned off in either process (or by yet another process) then it would be off in all processes. The default enforcement of the local predicate `No_Dangling_Children` is initially off in both `Proc_1` and `Proc_2`. `No_Dangling_Children` is turned on in `Proc_1` following the **`if`** statement. This affects the enforcement of `No_Dangling_Children` only in `Proc_1`. Prior to this point `Proc_1` can update `Bin_Tree` to violate `No_Dangling_Children` regardless of the enforcement of No_Dangling_Children in any other process. The default enforcement of No_Dangling_Children is also turned on at the beginning of Proc_2. Note that the enforcement setting contexts do not affect the default enforcement of the predicates but override it locally and temporarily. The default enforcement of Data_OK is off initially in Proc_2 and never turned on. Data_OK is not visible in Proc_1 and so cannot be enforced there.

Predicate "Enforced" Attributes The representation and control of the default enforcement of a predicate also depends on whether the predicate is global or local. A global predicate has a single boolean attribute enforced which determines whether or not it is enforced by default, and this single attribute is visible to all programs within the extent of the predicate. A global predicate can also be declared **enforced**, in which case the value of its enforced attribute is always true and cannot be changed. Such predicates are always enforced by default. A global predicate which is not declared enforced has an assignable enforced attribute. The default enforcement of such predicates can be changed by assignment to this attribute. In the example At_Most_One_Parent is a global enforced predicate. Thus, both Proc_1 and Proc_2 see a single value of the enforced attribute for At_Most_One_Parent. Since this predicate is not declared enforced that value may be changed, but both processes will see the results of any such change.

A local predicate has one boolean attribute enforced for each program in which it is used. Each such program sees a separate value for this attribute, and that value determines whether the predicate is enforced by default within that program. Local predicates cannot be declared enforced. The value of each enforced attribute is assignable by the program in which it is visible. In the example No_Dangling_Children is turned on by both processes by setting the local enforced attribute to true.

An issue which is not addressed in FCM is control over changes to the enforced attributes of predicates, i.e. control over the turning on and off of predicate enforcement. This is an especially important issue for global predicates, where changes to the enforced attribute may be made by any one of several affected (and possibly competing) processes. A specific approach to this problem is defined for APPL/A [23]; that approach is used in the examples. For example, No_Dangling_Children must be *acquired* before a value can be assigned to its enforced attribute. The acquire operation returns a key value (i in the examples) which is then given in the call to the enforced operation. This particular mechanism for setting enforced attributes is specific to APPL/A, but it illustrates the general idea. Other approaches are possible and could be instituted without affecting the basic concepts of the consistency model.

Two final comments on the example: First, the suspend statement is serializable (as described in Section 4.3). Thus there is no need for a separate serial statement where the suspend statement is used in the procedures. Second, the use of the suspend and enforce statements show how predicate enforcement can be controlled locally without manipulating predicate `enforced` attributes.

4.3 Control Constructs

As indicated in previous sections, individual operations on objects are assumed to be individually serializable and consistent with respect to enforced predicates. However, as in conventional databases, some mechanism is needed to group individual operations into composite operations which have transaction-like properties. This section defines five composite, more or less transaction-like statements that can be used to group other statements or operations. As a group these statements will be referred to as the consistency-management statements, or "CM" statements. The first subsection below provides an overview of the statements, including a design rationale. The second describes each statement individually. The third defines composition rules for the statements.

4.3.1 Overview

The CM statements introduced here are the *suspend*, *enforce*, *allow*, *serial*, and *atomic* statements. These are block statements which provide various combinations of serializability, atomicity (rollback), and control over predicate enforcement. Conventional transactions integrate serializability, atomicity, and suspension of constraint enforcement. However, with conventional transactions, there is no way to obtain any of these capabilities apart from the others. Section 2.3 presents arguments that software processes should benefit from relatively specialized control over serializability, atomicity, and predicate enforcement. The CM statements defined here are intended to enable such specialized control. The suspend, enforce, and allow statements are enforcement-setting, the suspend and atomic statements support rollback, and all but the enforce statement are serializable for operations on some set of objects. These capabilities are summarized in Table 4.3.1

It should be noted that some of the CM statements provide combinations of capabilities and that some capabilities are not available individually. For example, the suspend statement combines serializability, local suspension of designated predicates, and rollback, while the atomic statement combines serializability and rollback. On the other hand, no statement provides rollback or suspension of predicate enforcement without serializability. Thus the CM statements provide a middle ground between several statements with completely orthogonal capabilities and one construct that integrates all capabilities.

One might imagine (perhaps correctly) that the availability of completely orthogonal statements would further increase the flexibility of FCM. Certainly FCM could include a more orthogonal set of statements. For example, rollback and suspension of predicate enforcement could be provided without serializability. Thus the inclusion of statements which combine two or more capabilities deserves some comment.

Statement	Serializable	Enforcement-Setting	Atomic
Serial	yes	no	no
Suspend	yes	suspends[1]	yes[3]
Enforce	no	enforces[2]	no
Atomic	yes	no	yes[4]
Allow	yes	suspends[1]	no

Notes:

1. Suspends enforcement locally for designated predicates.
2. Requires enforcement locally for designated predicates.
3. Rollback for violation of enforced predicates on termination.
4. Rollback for propagation of an exception.

Table 1: Summary of Capabilities for FCM Consistency-Management Statements

Consider first the combination of serializability with rollback, which occurs in the suspend and atomic statements. The sharing of results that may be rolled back entails an element of risk. If one process depends on the results of another, and if those results are rolled back, then the results of the dependent process may be invalidated (and, consequently, need to be rolled back, perhaps necessitating a cascade of additional rollbacks). One might assume that the risk may be acceptable among "cooperating" transactions. However, any such premise of cooperation may be difficult to assure, i.e. processes that share data may actually "compete." Moreover, it may be impractical in general for a process to determine which data are shared with other processes and thus which data may be subject to rollback as a consequence of failure in other processes. A possible (and perhaps likely) response to this risk is for each process to protect itself by serializing it's own operations. This would reduce the possibility of rollback of shared data by reducing the sharing of data, but that is just the opposite of the desired effect.

The response to this problem in FCM is to require the serializability of any "transaction" that may be rolled back. This reduces the risk of invalidation (and cascading rollback) for other transactions. The requirement reduces sharing of the intermediate results of any transaction that has not finally committed. However, it promotes subsequent sharing of results once they are committed. Additionally, sharing may still take place among concurrent nested transactions within an enclosing transaction that may rolled back.

A similar argument applies to the association of serializability with the suspension of predicate enforcement in the suspend and allow statements. The suspension of predicate enforcement enables the creation of data that are at least temporarily inconsistent. This inconsistency could represent a risk to any process which shares the data but which requires them to be consistent. For a process that depends on consistent data the use of inconsistent data may lead to erroneous results or exceptions and rollback. As with the risk of rollback, a process may attempt to protect itself from this risk by adopting serializability, but the widespread use of this strategy may lead to an overall reduction of data sharing. The FCM approach is again to require the risk to be "encapsulated" by serializability. This reduces the adverse consequences of risk and may still enhance data sharing overall.

The suspend statement combines rollback with the suspension of predicate enforcement (in addition to serializability). As with conventional transactions, rollback is used to return to the previous (and presumed consistent) state in the event that the suspend does not satisfy enforced predicates. The allow statement does not entail rollback as a whole precisely because its purpose, unlike that of the suspend statement, is to enable the perpetuation of an inconsistent state. (This point is discussed with the allow statement.)

The one CM statement which does not entail serializability is the enforce statement. The enforce statement locally strengthens the consistency requirements for just the process in which it is used and it does not entail rollback. Consequently, it does not induce conditions that represent risks to other processes and from which other processes may need to be protected by serializability. (A second process may violate a condition that the first process locally enforces, but that violation adversely affects only the first process, and the first process can protect itself from this risk by nesting the enforce statement within a serial statement if necessary.) If a second process depends on the violation of a predicate then that process can use a suspend statement to obtain serializable access that allows but hides the inconsistency.)

In summary, the CM statements of FCM individually provide capabilities that are relatively specialized compared to conventional transactions. However, the CM statements do not represent individual orthogonal capabilities. Rollback and suspension of predicate enforcement are combined with serializability in order to encapsulate the risk associated with those capabilities. The suspend statement also includes rollback to assure the restoration of a previous state in the event of a predicate violation. Thus each of the CM statements in FCM can be viewed as providing a basic *safe* set of capabilities. More elaborate capabilities can be obtained by combining these statements (as described in Sections 4.4 and 5.1).

4.3.2 The Consistency Management Statements

This subsection describes each of five basic CM statements. The descriptions apply (for the most part) to basic cases in which the statements are not composed with one another (either by lexical nesting or dynamic calling). Rules for the composition of these statements are presented in the next subsection.

The Serial Statement Simple serializable access to objects can be obtained with the *serial* statement. The serial statement begins with a "read-write list" which designates objects to which serializable read or write access is desired. Read access is shared with other readers, write access implies read access and is exclusive. (Other kinds of access could be added; read and write access are included here as a simple minimum.) The naming of an object in a read-write list does not represent an access to the state of the object. However, it is a request for the indicated access right. Execution will proceed if and only if the indicated access right is obtained at that point; otherwise execution is blocked.

The serial statement does not affect the default or actual enforcement of predicates. Predicates are enforced in the scope of the serial statement as they are in the immediately-surrounding scope. 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.

Operations within the serial statement can refer to objects not designated in the read-write list. Access to these objects is obtained as requested. (Rules about serializability in this case are defined in Section 4.4.)

The Suspend Statement The *suspend* statement provides an enforcement-setting context in which the enforcement of designated predicates is suspended. The scope of suspension of enforcement of a designated predicate is the extent of the suspend statement, including any nested or called enforcement-setting contexts, but excluding any nested or called enforce statements in which the enforcement of the predicate is imposed.

The suspend statement provides serializable write access to the objects to which the suspended predicates apply. Write operations on these objects are logically *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.

The suspend statement does not affect the default enforcement of the predicates that are suspended but instead overrides it locally and temporarily. The suspend statement does not affect the actual (or default) enforcement of predicates that are not suspended. Any such predicates that are enforced in the surrounding scope are enforced within the suspend. The violation of an enforced predicate within the suspend results in the undoing of the violating operation and the raising of an exception. However, any exception raised within a suspend statement may be trapped, and in any case the propagation of an exception from a suspend statement does not necessarily entail rollback for the statement as a whole. The results of the suspend statement are rolled back if and only if the statement terminates in violation of an enforced predicate, regardless of whether the statement terminates normally or abnormally.

Within the suspend statement it is possible to refer to objects other than those referenced in the suspended predicates. Access to these objects is not serialized as a consequence of the suspend statement and must be obtained when called for. However, operations on these objects are logged by default and are rolled back if and when the suspend statement fails. This situation is described more fully in Section 4.4.

The Enforce Statement The *enforce* statement is another enforcement-setting context. In contrast to the suspend statement, however, it *imposes* rather than suspends the enforcement of designated predicates. The scope of imposition of enforcement of a designated predicate is the extent of the enforce statement, including any nested or called enforcement-setting contexts, but excluding any nested or called suspend statements in which the enforcement of the predicate is suspended. Any individual operation within the scope of the enforce statement that violates an enforced predicate is individually undone. Because each consistency-violating operation is undone individually as it occurs 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, and in order to promote shared access to data, the enforce statement is not serializable as a whole. (However, if desired, serializable access to objects can be obtained by nesting an enforce statement within a serial statement.)

Like the suspend statement, the enforce statement does not affect the enforcement of any predicate that is not explicitly designated. Within the enforce statement predicates that are not designated are enforced or not just as they are in the surrounding scope. Also like the suspend statement, the enforce statement does not affect the default enforcement of the designated predicates but overrides that default.

The Allow Statement The *allow* statement also creates an enforcement-setting context for designated predicates. Any designated predicate that is violated upon entry to the allow may be violated by operations within the allow (exclusive of nested or called enforce statements) *and also upon completion* of the allow. 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. As with the enforce statement, there is no rollback associated with the allow statement as a whole. However, the allow statement is serializable with respect to operations on objects referenced in the violated predicates. Serializability is imposed so that the inconsistent states which may hold within the allow are not visible to concurrent (but not otherwise serializable) processes that may require or attempt to establish consistency.

As with the suspend statement, operations within the allow statement may refer to objects to which serializable access is not initially obtained. Access to these objects is obtained as requested.

The Atomic Statement Serializable recoverable access to objects can be obtained with the *atomic* statement. Like the serial statement, the atomic statement has a read-write list of objects to which serializable read or write access is requested. As with the suspend statement, write operations on these objects are logically logged. Unlike the suspend statement the atomic statement does not affect predicate enforcement. Within the atomic statement predicates are enforced as they are in the immediately surrounding scope; the violation of an enforced predicate by an operation causes that operation to be undone but not any others. However, the propagation of an exception from the atomic statement does cause rollback of the entire results of the statement. The scope of rollback includes the logged operations performed within the extent of the atomic, including nested or called statements. (Rules about rollback for nested and called atomic statements are defined in the next section.) Because the atomic statement may entail rollback it is serializable.

4.4 Composition of the Statements

Both the consistency management statements defined in Section 4.3.2 and operations on data individually have semantics which may entail serializability, recoverability, and effects on predicate enforcement. The semantics of consistency management statements are defined above. Operations on data are presumed to be individual units of consistency and atomicity and also individually serializable.

The consistency-management statements and data operations are not restricted to individual use: they may be combined. The consistency management statements can be nested arbitrarily and can be used in operations on data. They may also occur in a subprogram called from another consistency management statement or from an operation on data. Additionally, one data operation may call another. The effects of such combinations of statements and operations must also be defined with respect to serializability, recoverability, and predicate enforcement. Rules for predicate enforcement in these cases are defined in Section 4.2. Rules regarding serializability and recovery are defined in this section.

For purposes of defining these rules the consistency management statements and data operations will collectively be referred to as consistency management *constructs*, or "CM constructs." The lexical or dynamic combination of these constructs will be referred to as their *composition*. A CM construct which nests or calls another CM construct is called an *including* construct; a CM construct which is nested within or called from another CM construct is called the *included* construct.

APPL/A includes two general approaches to the composition of CM-constructs:

- **Combined** composed constructs: the included constuct is functionally a subconstruct of the including construct with respect to serializability and recovery.

- **Separate** composed constructs: the included construct is functionally a separate construct from the including construct with respect to serializability and recovery.

In FCM the combined approach applies by default, but separate composed constructs can be designated explicitly. Combined and separate composed CM constructs are addressed in the next two subsections.

4.4.1 Combined Composed Constructs

The rules for serializability and recovery for combined composed CM constructs are enumerated below. These rules assume that none of the included constructs are "separate."

1. The serializable (recoverable) access to an object obtained on entry to a CM construct extends to all (possibly concurrent) included CM constructs. As a group these have serializable (recoverable) access with respect to any processes that are concurrent with the including construct. They compete with one another on an equal basis for serializable (recoverable) access within the including construct.

2. An included CM construct can obtain serializable (recoverable) access to an object for which serializable (recoverable) access is not held by an including CM construct. The serializable (recoverable) access associated with the included construct is acquired upon entry to that construct. The serializable (recoverable) access is held until the outermost including construct is completed. Such an included construct is treated like an access request on behalf of the including statement, and once that access is obtained it is held by the including construct.

3. An included CM construct can also obtain serializable (recoverable) access to an object for which serializable (recoverable) access is already held by an including CM construct. The serializable (recoverable) access associated with the included construct is acquired upon entry to that construct, at which point the serializable (recoverable) access held by the including construct is suspended. The serializable (recoverable) access held by the included construct is released when the construct is exited, at which point the serializable (recoverable) access held by the including construct is reinstated. The effect of included serializable (recoverable) access in this case is to achieve serializability (recoverability) with respect to operations that may occur concurrently within the including construct.

4. The rollback of an included CM construct takes place in accordance with rules for that construct. The rollback of an included construct does not directly entail the rollback of any including construct. An including atomic or suspend statement may still commit its results even when an included construct has rolled back. (However, the rollback of an included statement may be triggered by or associated with an exception which may in turn trigger the rollback of an including construct.)

5. The results of a recoverable CM construct which is included in another recoverable context are committed temporarily when that included construct successfully completes. These results are then visible within the including recoverable construct. However, they are not made visible outside of that including context until the context commits. Moreover, the rollback of the including context entails rollback of the results of included construct.

4.4.2 Separate Composed Constructs

Separate included consistency-management statements and individual operations are explicitly designated with the reserved word **separate**. Within a separate CM construct the rules for serializability and recovery are the same as they are for a combined CM construct. The relationship between the separate CM construct and the including CM construct is governed by the following rules:

1. The separate CM construct does not inherit any access from the including CM construct. It must obtain all access separately. It may share read access with the including construct (as with any other reading construct). Deadlock will result if the separate CM construct attempts to obtain serializable access which conflicts with access held by the including CM construct.

2. Serializable and recoverable access held by a separate CM construct is released when the CM construct terminates and is not assumed by the including CM construct. Results committed by a separate construct become visible at the time of commitment.

3. A separate CM construct commits or aborts its results upon termination independently of the including CM construct. In particular, the final commitment of the separate CM construct is not dependent on the final commitment of the including CM construct, and the separate CM construct may commit even if the including CM construct aborts.

A separate CM construct is thus logically and functionally a top-level CM construct even though it may be lexically nested in or called from another CM construct.

The general advantages of (combined) nested transactions (including concurrent nested transactions) are presented by Moss [13]. These include the subdivision of labor, encapsulation of potential failures, and opportunities for concurrency and distribution. Similar advantages extend to the more general combined composed transactions of FCM. In the context of software processes a combined composed transaction can be used to coordinate related or cooperating subprocesses. For example, several programmers working as a team can update various source modules concurrently as subtransactions included within an including maintenance transaction. Similarly, the work of a programmer and a unit tester can be coordinated as subtransactions within an including transaction which is intended to produce source code that satisfies certain functional or performance criteria.

The principal advantages of separate transactions are that they enable a single *lexical* transaction to commit various results independently and to acquire and release serializable access to objects differentially (thus promoting concurrency and data sharing). Separate transactions also provide a lexical mechanism for the synchronization of transactions that are functionally distinct. In the context of software processes, separate composed transactions can be used like combined composed transactions to coordinate subprocesses, including subprocesses that are not necessarily cooperative. For example, a single lexical transaction may represent the concurrent application of a suite of independent analysis tools and test programs to a source module. The quality of the input code may not be guaranteed, and any of the analyses or tests may abort. However, the failure of some analyses or tests should not preclude the extraction of results from the others or the retention of process logs from the main testing program. In this situation each analysis or test can be executed as a combined included transaction which which updates process logs as a part of the including transaction but which commits its results in separate subtransactions. An example along these lines is sketched (using APPL/A syntax) in Figures 4 and 5.

Separate composed transactions also have two special applications in software processes. One is the persistent logging of process activity, where the information logged is committed even if the including transaction is aborted. This information can be especially useful for process control and debugging. The second is in the "savepointing" of the intermediate results of a transaction so that not all of the work of the transaction is lost in the event of rollback. The ability to preserve intermediate results even in the event of rollback is especially important in software processes (and related design activities) where transactions may be very long and difficult to redo. While FCM does not explicitly include a savepoint facility, separate composed transactions provide a mechanism by which explicit savepoint operations can be programmed in any particular process as appropriate.

5 Examples

This section contains several additional examples to illustrate the ways in which various features in FCM can be used. The first subsection illustrates combinations of CM statements which achieve the effects of other conventional and unconventional constructs. These include conventional "flat" transactions, conventional "nested" transactions, "assertion" transactions, and a "repair-enforce" statement. The following three subsections present sketches of simple software-process programs. The first of these illustrates centralized, managerial control of consistency maintenance in the context of a requirements-specification process. The second illustrates decentralized, federated control of consistency maintenance among a group of co-equal processes which share data. The third illustrates cooperative work among a team of programmers who work together on coding releases of source-code modules.

The process programs, as noted, are simple sketches. They are intended primarily to illustrate the features of FCM in a relevant context. For the most part the details of the software processes are irrelevant and so are omitted. The process programs are coded (to the extent that they are coded at all) in an APPL/A-like syntax. APPL/A is an extension of Ada, so these programs generally resemble Ada programs (they include, for example, Ada tasks and exception handlers). Consistent with APPL/A, the important data in the programs are stored in relations (which are barely sketched), and the APPL/A syntax for predicates and the CM statements is used.

```
-- testing and analysis procedures:
with Execution_Timer, Data_Flow_Analyzer, Exception_Flow_Analyzer, ...;
-- data repositories for test and analysis results:
with Execution_Times, Data_Flow_Analyses, Exception_Flow_Analyses, ...;
with Exception_Log; -- data repository for exception reports
with Test_Log; -- data repository for status of tests

procedure Test_and_Analysis_Suite(src:  in source_code) is
begin
    atomic write Test_Log, Exception_Log;
    begin
        loop
            -- evaluate test and exception logs to determine
            -- status of process
            serial read Test_Log, Exception_Log;
                ...
            end serial;

            if done then
                exit;
            elsif timing_next then
                do_timing(src, get_data);
            elsif ...        -- do other tests or analyses
                ...
            end if;
        end loop;
    exception
        -- Depending on exception and status, either handle and
        -- commit results in Exception_Log and Test_Log or propagate
        -- and rollback results in Exception_Log and Test_Log
        ...
    end atomic;
end Test_and_Analysis_Suite;
```

Figure 4: Sketch of Procedure Test_and_Analysis_Suite

```
procedure Do_Timing(src, data) is
    execution_time:  time_type;
begin
    -- run test
    Get_Execution_Time(src, data, execution_time, status);
    -- save test results separately
    separate Execution_Times.update(src, execution_time);
    -- update test log with test status
    Test_Log.update(‘‘timing’’, src, data, status);
exception
    when others =>
        -- update exception log and test log
        -- but do not propagate exception
        Exception_Log.update(...);
        Test_Log.update(‘‘timing’’, src, data, status(‘‘exception’’));
end do_timing;
```

Figure 5: Testing Procedure with a Separate Included Transaction

5.1 Some Combinations of Consistency Management Statements

The consistency management statements introduced in Section 4.3 can be combined in various ways to achieve various effects. Some of these are conventional; others are not. Some useful combinations of CM statements are described in the following paragraphs.

Conventional Flat Transactions A suspend statement for predicates that apply to a given object can be nested within an atomic statement that writes to that object to achieve the effects of a conventional transaction, i.e. a serializable, consistent, and complete unit of work with respect to the object. An example is sketched in Figure 6.

Conventional Nested Transactions Several suspend statements that apply to an object can be nested sequentially and/or concurrently within an atomic statement that writes that object to achieve the effects of the nested concurrent transactions of Moss [13]. A simple example without concurrency is sketched in Figure 7; this example makes use of "combined" subtransactions in FCM.

"Assertion" Transactions An enforce statement for predicates that apply to an object can be nested within an atomic statement that writes that object to achieve an effect com-

```
atomic write O;         -- object O
begin
    suspend P;          -- predicate P applies to O
    begin
        -- operate on O here, unconstrained by P
        ...
    end suspend;        -- check P:  if violated, then
                        -- rollback operations in suspend
                        -- and raise Constraint_Error
end atomic;     -- rollback if any exception is propagated
                -- (Constraint_Error or otherwise)
```

Figure 6: Sketch of a "Conventional Flat Transaction" in FCM

plementary to conventional transactions with respect to predicate enforcement. An example is sketched in Figure 8. This construct might be called an "assertion" transaction because the enforced predicate becomes an assertion on the transaction. The violation of that predicate by any individual operation within the enforce statement raises Constraint_Error. That exception indirectly causes the whole construct to be rolled back (when the exception is propagated through the atomic statement).

"Repair-Enforce" Statements An enforce statement for a predicate can be nested within the latter part of a serial statement that obtains serial write access to the objects referenced by an unenforced predicate to achieve the effects of the "repair-enforce" statement proposed in [23]. An example is sketched in Figure 9. In this construction, the serial statement assures serializable access to the designated object; this access can be used to "repair" predicate violations free of interference from other processes. Following the repair operations the enforce statement assures continued satisfaction of required predicates. Note that there is no rollback for this construct as a whole; that capability could be obtained by further nesting this construct within an atomic (as in Figure 8).

These are just a few examples of the combination of CM statements. Many other combinations are possible.

5.2 Managerial Control of Consistency Maintenance

This section illustrates the managerial control of consistency maintenance in a sketch of a simple software-process program for the development of software requirements. In this

```
atomic write O;         -- object O
begin
    suspend P;          -- predicate P applies to O
    begin
        -- operate on O here, unconstrained by P
        ...
    end suspend;        -- check P:  if violated, then
                        -- rollback operations in suspend
                        -- and raise Constraint_Error

    -- operate on O here, but constrained by P
    ...

    suspend P;          -- predicate P applies to O
    begin
        -- operate on O here, unconstrained by P
        ...

    end suspend;        -- check P:  if violated, then
                        -- rollback operations in suspend
                        -- and raise Constraint_Error

end atomic;     -- rollback if any exception is propagated
                -- (Constraint_Error or otherwise)
```

Figure 7: Sketch of a “Nested Transaction” in FCM

```
atomic write O;         -- object O
begin
    enforce P;          -- predicate P applies to O
    begin
        -- operate on O here, constrained by P:
        -- any operation that violates P will be undone
        -- and cause the raising of Constraint_Error
        ...
    end enforce;
end atomic;     -- rollback if any exception is propagated
                -- (Constraint_Error or otherwise)
```

Figure 8: Sketch of an "Assertion" Transaction in FCM

```
serial write O;         -- object O
begin
    -- operate on O here to repair violations of predicate P
    -- (P applies to O but is not enforced)
    ...

    enforce P;          -- P now locally enforced
    begin
        -- operate on O here, constrained by P:
        -- any operation that violates P will be undone
        -- and cause the raising of Constraint_Error
        ...
    end enforce;
end serial;
```

Figure 9: Sketch of a "Repair-Enforce Statement"

approach to consistency maintenance a manager controls both the activities of subordinates and also the enforcement of predicates on the data on which the subordinates work.

The requirements process is sketched in APPL/A-like syntax. The requirements take the form of a graph of requirements elements. The graph represents a functional decomposition of the elements. The graph is modeled by two APPL/A relations (sketched in Figure 10). Relation `Requirements_Elements` stores tuples which correspond to individual requirements. Relation `Requirements_Graph` stores tuples which indicate edges between the elements. (The details of the elements and the graph are largely irrelevant to the example and so have been omitted.) The consistency of the requirements graph is represented by two general predicates over the relations. Predicate `Req_Elements_OK` tests acceptance criteria for the requirements elements stored in `Requirements_Elements`. Predicate `Req_Graph_OK` tests acceptance criteria for the edges of the graph stored in `Requirements_Graph`. (Again, for purposes of this example, the details of the predicates are irrelevant.)

The managerial role in the requirements process is played by the procedure Requirements_Manager (Figures 12 and 13). This simplified software process has two principal functions: the invocation of subprocesses which perform the actual work on the requirements, and control over enforcement of the predicates that apply to those requirements. There are three subprocesses (sketched in Figure 11):

- Procedure `Identify_Elements` identifies individual requirements and stores them in `Requirements_Elements`. While the elements are being created the predicate `Req_Elements_OK` is temporarily violated, but the procedure (if successful) leaves each element in a state that is consistent with the predicate.

- Procedure `Build_Graph` constructs edges for the requirements graph based on elements stored in `Requirements_Elements`. It depends for its correct execution on the satisfaction of `Req_Elements_OK`, but it temporarily violates `Req_Graph_OK`. If it is successful, however, it will leave `Req_Graph_OK` satisfied.

- Procedure `Analyze_Graph` analyzes the requirements graph for completeness and consistency. It does not update the relations, and so does not affect their consistency, but it depends on the satisfaction of both predicates. It stores its results in the relation `Analysis_Results`.

Note that none of these processes manipulates the predicates.

The managerial process `Requirements_Manager` first acquires the two predicates so that it can set their enforced attributes and thereby control their enforcement. The main part of the procedure is a loop over a case statement, where the branches are selected by values of the variable `next_phase`. The phases of the process include `identify`, `build`, `analyze`, and `done`. For each branch, the procedure sets the enforcement of the predicates to meet the assumptions of the corresponding subprocedure and then invokes that subprocedure. The initial value of next_phase is identify. In the "identify" branch the default enforcement of both predicates is turned off and the subprocedure Identify_Elements is invoked. If the subprocedure leaves Req_Elements_OK satisfied then next_phase is set to build. In the "build" branch the default enforcement of Req_Elements_OK is turned on, while that of Req_Graph_OK is left off, and the subprocedure Build_Graph is invoked. If Build_Graph leaves both predicates satisfied then next_phase is set to analyze. In the "analyze" branch the default enforcement of Req_Graph_OK is also turned on and the subprocedure Analyze_Graph is invoked. The results stored in Analysis_Results are evaluated to determine the next phase, which may be identify, build, or done. Thus, if the next phase is not done, the process is reiterated.

The software process represented by Requirements_Manager is admittedly simplistic, but it illustrates the general approach to managerial control of consistency maintenance for subordinates. The subordinate processes are coded assuming that they will be executed under the correct conditions of consistency enforcement. In this respect they are simplified because they do not have to include code to assure those conditions themselves. In this example the control of consistency maintenance was achieved by setting the default enforcement of predicates; it could also have been achieved by invoking the subprocedures within suspend and enforce statements that provided the appropriate enforcement-setting context.

5.3 Federated Control of Consistency Maintenance

This section illustrates control of consistency maintenance in a system in which several coequal processes share access to data. These processes are cooperative in that they must work in concert to achieve a common goal; however, their short-term individual operations on the data may often conflict. Thus, they share data according to an agreed-upon protocol that enables cooperation while avoiding conflicts.

The scenario for this example includes top-level software processes for the development of requirements, designs, and code. Individually, each of these processes can be regarded as a "manager" for several subprocesses, similar to the requirements-manager procedure described in Section 5.2. However, the details of the subprocedures are unimportant for this example and so have been omitted. The manager processes share project management data that are stored in a relation Project_Data. A mandatory predicate Project_Data_OK represents the consistency of the relation Project_Data. (The details of the relation and predicate are also irrelevant to the example.) A sketch of these elements is shown in Figure 14.

The top level managerial processes for requirements, design, and coding operate concurrently and in a similar way. Each manager process orchestrates its work based on the data in Project_Data, performs that work, and then updates Project_Data accordingly. Although the predicate Project_Data_OK is mandatory it is not enforced at the time these procedures are executed and each may update Project_Data in such a way

```
Relation Requirements_Elements is
-- Tuples correspond to individual requirements elements
    ...
End Requirements_Elements;


Relation Requirements_Graph is
-- Tuples correspond to edges between requirements elements
    ...
End Requirements_Graph;

        --------------------------------

global predicate Req_Elements_OK is
-- Tests acceptance criteria for requirements elements
-- stored in Requirements_Elements
    ...
End Req_Elements_OK;


global predicate Req_Graph_OK is
-- Tests acceptance criteria for requirements graph structure
-- stored in Requirements_Graph
    ...
End Req_Graph_OK;
```

Figure 10: Sketch of Relations and Predicates for Requirements Process

```
with Requirements_Elements;
Procedure Identify_Elements is
-- Identifies requirements and stores them in Requirements_Elements.
-- Must violate the predicate Req_Elements_OK while elements are
-- being created, but leaves that predicate satisfied.
Begin
    ...    -- No manipulation of predicates
End Identify_Elements;

        --------------------------------

with Requirements_Elements, Requirements_Graph;
Procedure Build_Graph is
-- Reads requirements elements in Requirements_Elements and builds
-- a graph of them with edges stored in Requirements_Edges.  Depends
-- on Requirements_Elements satisfying Req_Elements_OK.  Must
-- violate the predicate Req_Graph_OK during graph construction,
-- but leaves it satisfied.
Begin
    ...    -- No manipulation of predicates
End Build_Graph;

        --------------------------------

with Requirements_Elements, Requirements_Graph;
with Analysis_Results;  -- stores analysis results
Procedure Analyze_Graph is
-- Analyzes the requirements graph (represented by Requirements_Elements
-- and Requirements_Edges) for desired properties.  Stores results in
-- Analyze_Results.  Depends on the satisfaction of both Req_Elements_OK
-- and Req_Graph_OK.
Begin
    ...    -- No manipulation of predicates
End Analyze_Graph;
```

Figure 11: Sketch of Subprocedures for Requirements Process

```
with Requirements_Elements, Requirements_Graph,    -- requirements relations
     Req_Elements_OK, Req_Graph_OK,         -- predicates on those relations
     Identify_Elements, Build_Graph, Analyze_Graph,          -- subprocesses
     Analysis_Results;                    -- stores results of Analyze_Graph
procedure Requirements_Manager is
-- Invokes subprocesses for the identification of requirements elements,
-- building of the requirements graph, and analysis of the requirements
-- graph.  Controls the enforcement of predicates as needed to assure
-- proper degree of consistency maintenance for those subprocesses.
--
    i,j:  integer
    type process_phase = (identify, build, analyze, done);
    next_phase:  process_phase := identify;
    element_error, graph_error, analysis_error:  exception;
Begin
    -- Acquire predicates on the relations in which the
    -- requirements elements and graph are stored
    i := Req_Elements_OK'acquire;
    j := Req_Graph_OK'acquire;

    while next_phase /= done loop
        case next_phase is
            when identify =>    -- Identify requirements elements
                -- Turn enforcement of both predicates off
                Req_Elements_OK'enforced(i, false);
                Req_Graph_OK'enforced(j, false);

                -- Invoke the subprocess
                Identify_Elements;

                -- Check that elements are OK for graph building
                if Req_Elements_OK then
                    next_phase := build;
                else
                    raise element_error;
                end if;
```

Figure 12: Sketch of Requirements_Manager Procedure – Part 1

```
            when build =>    -- build requirements graph
                -- Set predicates
                Req_Elements_OK'enforced(i, true);
                Req_Graph_OK'enforced(j, false);

                -- Invoke the subprocess
                Build_Graph;

                -- Check that graph is OK for analysis
                -- (elements should still be OK since
                -- Req_Elements_OK has been enforced)
                if Req_Graph_OK then
                    next_phase := analyze;
                else
                    raise graph_error;
                end if;

            when analyze =>    -- analyze requirements graph
                -- Turn enforcement of both predicates on
                Req_Elements_OK'enforced(i, true);
                Req_Graph_OK'enforced(j, true);

                -- Invoke the subprocess
                Analyze_Graph;
                next_phase := evaluate(Analysis_Results);

            when done =>
                null;
        end case;
    end loop;
End Requirements_Manager;
```

Figure 13: Sketch of Requirements_Manager Procedure – Part 2

```
-- relation to store management data shared by concurrent
-- requirements, design and coding software processes
relation Project_Data is ... end Project Data;

-- predicate on relation Project_Data
mandatory predicate Project_Data_OK is ... end;

--
-- procedures that use relation Project_Data
--
procedure Requirements_Manager is ... end;

procedure Design_Manager is ... end;

procedure Code_Manager is ... end;
```

Figure 14: Sketch of Relations, Predicates, and Procedures for Federated Software Processes

as to violate Project_Data_OK. However, each process may also occasionally require that Project_Data_OK be satisfied when the process is reading or updating Project_Data.

The sketch of a "generic" body for these managerial procedures is shown in Figures 15 and 16. Each body consists of a loop with three principal parts: a reading of Project_Data to plan the work to be done, the work itself, and an updating of Project_Data to reflect that work. The initial reading of Project_Data is performed in a serial read statement; this allows any of the other processes to read the relation but prevents concurrent updates which would interfere with that reading. If the evaluation of Project_Data depends on the satisfaction of Project_Data_OK but the predicate is not satisfied then an attempt is made to "repair" the relation. The repair action is performed in a nested serial write statement. Following the basic work of the procedure (requirements analysis, design, or coding) the project data are updated in a way that depends on whether Project_Data_OK must be satisfied. If not, then the procedure simply updates Project_Data directly, without obtaining serializable access or enforcing the predicate. Other processes may concurrently read and/or update the relation (if these activities aren't themselves serialized). Interference with concurrent processes which require serializable access or consistent data is not a problem because those processes will have obtained those conditions as necessary (thus preventing the given process from performing the updates and violating the predicate). If the satisfaction of Project_Data_OK is required, then the procedure obtains serializable write access to Project_Data, repairs the relation as necessary, and then updates the relation using an enforce statement which assures continued satisfaction of the predicate. Because access to the relation is serializable no other process can interfere with the update and cause the predicate to be violated.

In this example none of the top-level managerial processes has global control over consistency maintenance or access to the shared data. Moreover, each process may have different expectations for consistency maintenance on different iterations. To accommodate this situation each process controls the serializability and consistency maintenance of its own activities. Serializability in particular offers protection from interference by other processes. Consistency is maintained through the use of CM statements, as compared with the manipulation of predicates in the example of Section 5.2. In this example the CM statements reflect directly and simply the processes' requirements for predicate enforcement; however, operations on predicates could also have been used.

5.4 Cooperative Work

This section illustrates how the CM statements and separate included transactions can be used in a program of cooperative work. The example involves a group of programmers who work cooperatively in the coding of a set of source modules.

The main procedure, `Cooperative_Coding`, is sketched in Figure 17. The procedure imports a relation type, `Source_Repository`, which defines stores for source modules. It also imports a specific instance of this type, `Release_Repository`, which serves as a global store for released modules. The procedure declares a second instance of this type, `Team_Repository`, which serves as a local shared store of modules for the programming team. The procedure also declares a task type which represents the programmers.

The body of the procedure consists of a loop principally over an atomic statement. The atomic statement writes the relation `Team_Repository`. Thus operations on that repository are treated somewhat like transactions. If the atomic statement is terminated by the propagation of an exception, for example `Project_Canceled`, the intermediate results of the atomic statement are rolled back. If the atomic statement terminates normally then the results in Team_Repository are committed. Because the atomic statement is in a loop, it can be executed repeatedly for various projects (or repeated attempts on the same project). An array of programmer tasks is declared within the atomic statement. The procedure `Cooperative_Coding` assigns various activities to these tasks, including coding new modules, updating existing modules, and copying released modules into the global relation `Release_Repository`.

The body of the programmer task type is sketched in Figures 18 and 19. The task declares a relation `My_Modules`, of type `Source_Repository`, to serve as a local workspace. The body of the task is largely a loop over a case statement. The branches of the case statement correspond to the activities of creating, updating, and releasing modules. In the

```
Begin
    main:  loop
        read_loop:  loop
            -- Serially read project data to plan work
            serial read Project_Data;
            begin
                if not Project_Data_OK and need(Project_Data_OK) then
                    serial write Project_Data;
                        if possible then
                            repair_project_data;
                            status := read_data;
                        else
                            status := try_again;
                        end if;
                    end serial;
                end if;
                if status = read_data then
                    -- read project data here; update status accordingly
                    ...
                end if;
            end serial;
            if status = try_again then
                delay some_duration;    -- and repeat read loop
            else
                exit;                   -- read loop
            end if;
        end loop read_loop;
        exit when status = done;    -- main loop
```

Figure 15: Sketch of Typical Procedure Body for Federated Software Processes – Part 1

```
        -- Do work on project depending on status
        ...

        -- Update project data to reflect work accomplished
        if don't_need(Project_Data_OK) then
            -- update project data in a way that may
            --     violate Project_Data_OK
            -- don't use serializable access
            ...
        elsif need(Project_Data_OK) then
            -- get serializable access to help
            -- assure consistency
            serial write Project_Data;
            begin
                if not Project_Data_OK then
                    -- repair Project_Data if possible
                    ...
                end if;

                enforce Project_Data_OK;
                begin
                    -- update project data in a way that
                    --     satisfies Project_Data_OK
                    ...
                end enforce;
            end serial;
        end if;
    end loop main;
End;
```

Figure 16: Sketch of Typical Procedure Body for Federated Software Processes – Part 2

```
with Source_Repository;    -- relation type to store source modules
with Release_Repository;   -- global repository of released source
                           -- modules (instance of Source_Repository)
procedure Cooperative_Coding is
    type activity = (create_modules, update_modules, release_modules);
    Project_Canceled:  exception;

    Team_Repository:  Source_Repository;    -- shared store of source
                                            -- modules for the team
    task type Programmer;
    task body Programmer is separate;
begin
    loop
        get_project_specifications;
        atomic write Team_Repository;
            Programmer_Team:  array(1..team_size) of Programmer;
        begin
            -- assign coding tasks to different programmers:
            --   - create new modules in Team_Repository
            --   - update existing modules in Team_Repository
            --   - release some set of modules to Release_Repository
            ...
        exception
            when project_canceled => raise;
                -- Propagation of an exception causes uncommitted
                -- modules in Team_Repository to be rolled back;
                -- modules in Release_Repository are not affected.
            when others =>
                ...
        end atomic;

        -- assess project status
        ...
        exit when done;
    end loop;
end Cooperative_Coding;
```

Figure 17: Sketch of Procedure Cooperative_Coding

"create" branch, new modules are stored temporarily in My_Modules. When complete, these are copied in an atomic statement to Team_Repository. The atomic statement assures that the copying will be an all-or-nothing activity. Moreover, it will be serializable, so no other programmer can see an incompletely copied set of modules. In the "update" branch the modules to be updated are read from Team_Repository into My_Modules. The reading takes place within a serial statement that assures that no other programmer can update or delete the modules while they are being copied. Once the modules are updated, they are copied back into Team_Repository in an atomic statement. In the "release" branch the modules to be released are copied from Team_Repository to Release_Repository. Team_Repository is read serially to prevent interference by other processes. Release_Repository is written in a nested "separate" atomic statement. Because the writing of Release_Repository is in an atomic statement it is an all-or-nothing, serializable operation. Because that atomic statement is separate the results of the atomic are committed independently of other results of the task. Thus, even if the programmer task or main procedure fails because of an unanticipated exception (say an operating error), the released results will persist and the work they represent will not be lost.

In contrast to the previous examples, this example focuses on the coordination of cooperating workers. It illustrates several features that may be associated with data management in this kind of activity. First, it shows that interactions involving shared data (in Team_Repository) may be relatively uncontrolled. Certain reading and writing activities are secure in the sense that they are serialized, but there are few other explicit safeguards on programmer interactions. Thus the example represents a case in which the success of the activity is premised on true cooperation among participants largely unconstrained by the process program. (Of course stronger controls on programmer interaction could be programmed if desired.) Second, it shows the use of workspaces at three levels: individual programmers have their own repositories which are hidden from other programmers, the procedure Cooperative_Coding has its own workspace which is shared by the programmers but hidden from the outside world, and the relation Released_Modules represents a repository which is globally accessible. Finally, the example shows various approaches to transferring data between these repositories, including serialized access, atomic access, and "separate" subtransactions.

6 Discussion

This section presents discussions of several issues in the design and use of the consistency and transaction models of FCM. These include a comparison of the model with the requirements, the introduction and management of inconsistency, and serializability and deadlock.

```
separate (Cooperative_Coding)
task body Programmer is
    My_Modules:  Source_Repository;  -- local store for the programmer
begin
    loop
        activity := get_next_activity;
        case activity is
            when create_modules =>
                -- get specifications for new modules
                ...
                -- create new modules and store temporarily
                -- in My_Modules
                ...
                -- copy new modules into Team_Repository
                atomic write Team_Repository;
                begin
                    for t in My_Repository loop
                        Team_Repository.insert(t);
                    end loop;
                end atomic;
                -- delete local copies of modules
                ...
            when update_modules =>
                -- get specifications for modules to update
                ...
                -- copy these modules from Team_Repository into
                -- My_Modules
                serial read Team_Repository;
                begin
                    for t in Team_Repository where
                        satisfies(t, update_specification)
                    loop
                        My_Modules.insert(t);
                    end loop;
                end serial;
```

Figure 18: Sketch of Programmer Task Body – Part 1

```
                -- update the modules
                ...
                -- copy updated modules back into Team_Repository
                atomic write Team_Repository;
                begin
                    for t in My_Repository loop
                        Team_Repository.insert(t);
                    end loop;
                end atomic;
                -- delete local copies of modules
                ...
            when release_modules =>
                -- get specifications for modules to release
                ...
                -- copy specified modules from Team_Repository
                -- to Release_Repository
                serial read Team_Repository;
                begin
                    separate atomic write Release_Repository;
                    begin
                        for t in Team_Repository where
                            satisfies(t, release_specification)
                        loop
                            Release_Repository.insert(t);
                        end loop;
                    end atomic;
                end serial;
            end case;
        exit when done;
    end loop;
end Programmer;
```

Figure 19: Sketch of Programmer Task Body – Part 2

6.1 Comparison of FCM and the Requirements

FCM directly addresses the requirements set forth in Section 3. The ability to turn predicates on and off is provided by two mechanisms. Assignable `enforced` attributes for predicates enable control over the default enforcement of predicates within the enforcement scope of the predicate (i.e. either collectively or individually for global or local predicates, respectively). The enforcement-setting suspend, allow, and enforce statements provide direct control over actual enforcement within their scope. These two mechanisms, in combination with the ability to declare predicates **enforced**, also provide global and local controls over predicate enforcement.

The control constructs provide a high degree of flexibility in functionality of the kinds required. The serial, suspend, atomic, and allow statements provide serializable access; the serial statement provides serializable access only, while the others provide it in conjunction with other capabilities (such as rollback) for which serializable access seems necessary (see Section 4.3.1). The suspend and allow statements provide local escape from predicate enforcement, while the enforce statement supports the local imposition of predicate enforcement. The suspend and atomic statements provide atomicity, the suspend for consistency of work, the atomic for completeness of work. The consistency management statements can also be combined in various ways to achieve additional functionality (see Section 5.1).

6.2 Inconsistency

The consistency model of conventional transactions is intended to assure that no transaction leaves the data in an inconsistent state. In effect this model prohibits constraint-violating operations on data. Balzer [3] has proposed a consistency model in which inconsistency is accommodated. In this model inconsistency (apparently) arises directly as a result of predicate-violating operations on data (see Section 7). Predicate violations must be repaired by transactions that can operate in the inconsistent state.

FCM also accommodates inconsistency. However, in analogy with the conventional model, no CM statement is allowed to leave the data in violation of an enforced predicate. (The one exception is the allow statement, the purpose of which is precisely to enable an *existing* violation to be perpetuated.) Instead, inconsistency arises when a new predicate is enforced on existing data. [1] Thus all CM statements (apart from the allow) are required to preserve consistency, but the criteria for consistency can change over time. Repairs to a violated predicate are performed by suspending the enforcement of the predicate, either temporarily and locally (using a suspend or allow statement), or indefinitely and possibly globally (by setting it's `enforced` attribute to false).

It should be noted that in both of these models the criteria for consistency are explicit in

[1] Inconsistency can also arise when one process violates a local predicate that is enforced in other processes, but those other processes are still not permitted to violate that predicate themselves.

the form of constraints or enforced predicates. Moreover, these criteria can be tested so that the consistency of a state can be evaluated. In other models, for example HiPAC [12] and the model of Sarkar [21], some constraints are not stated explicitly but are enforced implicitly by triggers. In such systems the criteria for consistency must be inferred, inconsistency will occur before the triggered actions can take effect, and the actual consistency of a state of state can be difficult to assess. These systems may be said to harbor inconsistency but not to accommodate it very effectively.

6.3 Serializability and Deadlock

In a transaction model in which objects can be locked and unlocked in arbitrary sequence some legal schedules of operations may not be serializable. To ensure the serializability of legal transaction schedules a *two-phase* lock/unlock protocol can be used. In this protocol a transaction must perform all of its lock operations before performing any unlock operations. If this protocol is used then the serializability of transactions is guaranteed. (The serializability problem and two-phase protocol are discussed in standard database textbooks, for example [25].)

The design of the serializable CM statements in FCM ensures that the two-phase protocol is observed. Each individual operation is serializable, and each individual serializable CM statement is a block statement in which serializable access is obtained on entry and released on exit. The two-phase protocol is preserved by the composition rules for combined and separate composed transactions. In a combined composed transaction, all serializable access, once acquired, is held until the execution of the transaction terminates. A similar rule applies within a separate composed transaction, which internally is treated like a combined composed transaction. Because a separate subtransaction is logically a distinct transaction from the including transaction (even though one may be lexically nested in the other) the separate subtransaction cannot interfere with the protocol for the including transaction (and vice versa).

The two-phase protocol does not, in and of itself, prevent deadlock among legal serializable transaction schedules. There are three general approaches to the problem of deadlock:

- Prevent deadlock by requiring transactions to be coded in such a way that deadlock cannot arise. This can be done by requiring access to objects to be obtained in a canonical order. This approach depends on methodological and analytic support to assure that transactions are properly coded.

- Avoid deadlock by preventing the concurrent execution of transactions that may deadlock. This requires analytic support to determine the access requirements of transactions and runtime support to monitor and control the execution of transactions.

- Break deadlocks that occur by rolling back one or more of the deadlocked transactions. This requires runtime support for the detection and breaking of deadlock.

The last of these is usually regarded as unacceptable in software environments (and other environments with long transactions) because rollback may entail the loss of large amounts of work. Consequently, many advanced transaction models assume (implicitly or explicitly) that serializability is obtained without deadlock.

In FCM the possibility of deadlock is admitted as the cost of certain kinds of flexibility. In particular, the composition rules for serializable statements in FCM allow serializable access to be requested only as needed, not necessarily at the outermost level of a transaction. By default, in a combined included transaction, as in conventional nested transactions, that access is held until the transaction terminates. With the use of a separate included transaction the serializable access is still obtained if and when requested, but it is released at the end of the included transaction, not the including transaction. Both approaches enable serializable access obtained by the included transaction to be held for only part of the execution time of the including transaction.

This additional flexibility is not enough of a benefit to justify the potential for arbitrary deadlock if deadlock may entail rollback and the consequent loss of large amounts of work. However, FCM admits several approaches whereby the potential for deadlock or loss of information can be reduced or eliminated. These include programming conventions to prevent deadlock or avoid costly rollback, runtime control over transaction execution, and reduction of the need for serializable concurrent access. Each of these is discussed below.

The transaction model of FCM allows serializable access to data to be programmed in conventional ways that avoid deadlock or minimize its impact. Specifically, it is possible to write transactions which which acquire serializable access to objects in a canonical order, or which acquire serializable access to all needed objects before any work is done. Two examples of this approach:

1. An atomic statement can be written to acquire serializable access to all referenced objects at the outermost level. Included statements do not obtain serializable access to any other objects. In this case the atomic statement proceeds if and only if it has the required access. If that access cannot be obtained, the statement can be aborted and retried without loss of work. If the access is obtained initially, then the statement can execute deadlock-free.

2. A nested serial statement can be written so that serializable access is obtained in a sequence of (nested) steps before any other work is done. If access cannot be obtained then the whole nested statement is rolled back, but no work is lost. If all requested access is obtained then the nested statements can execute deadlock-free. This is comparable to the above example, except that here the serializable access held by the inner statements can be released within the scope of including statements before the including statements terminate.

In this approach the full flexibility of FCM is curtailed, but that flexibility is still available for situations in which deadlock is not expected to be a problem. This approach does depend on programming conventions, and it may be difficult to determine in advance which objects may be referenced by a transaction (particularly if those references are determined dynamically). However, conventions such as these can be supported and enforced by standardized development methodologies, analysis tools, and runtime controls. Such features are expected to be integral to software-process programming environments, such as Arcadia [24], for which FCM is intended.

In the absence of programming conventions to avoid deadlock, deadlock can still be prevented by runtime control over the execution of programs that use serializable statements. Analysis of such programs can provide information on the objects to which serializable access is (or may be) requested and the order in which that access may be acquired and released. At the time a transaction is started a check can be made to determine whether it may deadlock with any ongoing transactions. If so, the execution of the transaction can be delayed, or the initiator of the transaction can be warned. The status of ongoing transactions can be monitored, and pending transactions can be initiated as soon as there is no potential for deadlock. In long-duration transactions the time spent in analyzing programs and monitoring execution will be comparatively minor.

The above solutions apply to cases in which objects are shared by transactions. Shared access can require serializability, and serializability can entail execution delays. Those delays will be proportional to the duration of transactions, and in a software environment the transactions, and thus the delays, may be very long. The delays can be reduced to the extent that the need for serializability can be reduced. FCM explicitly includes two approachs to reducing the need for serializable access and implicitly accommodates two others.

FCM provides two general capabilities which allow the extent of serializable access to be closely controlled and thus minimized. First is the capability to control when and where predicates are enforced. Serializability is required whenever an enforced predicate is suspended, but the enforcement of predicates can be limited to just those times and programs where they are relevant. Second is the capability to control the extent over which serializable access is held. This is provided by the flexible composition rules for the consistency management statements, especially separate transactions. These enable serializable access for some objects to be held for a (possibly small) portion of the time it is held for other objects.

The two capabilities which FCM implicitly accommodates relate to automatic savepoints and persistent versions. Savepoints preserve intermediate results in the event that a transaction is rolled back, thus reducing the cost of rollback in terms of work lost. Persistent versions are used to give conflicting concurrent transactions different "copies" of a shared object, thus enabling them to proceed in parallel [26,12]. A related approach is long-term transactions with check-out and check-in, in which users make their own copies of an object [21,9]. Neither savepoints nor copying/versioning is explicitly included in FCM. However, it is assumed that FCM is embedded in a process programming language with general programming capabilities (for example, APPL/A [23]). These capabilities can be applied manually to save data and to make and merge copies, thus achieving the principal effects of savepoints persistent versions, and check-out/check-in. A process programming language may also provide automatic support for these features on top of FCM. Separate included transactions may be especially useful in this regard.

In summary, the potential for deadlock arising from serializability in FCM can be reduced or eliminated in three ways:

- enforcible conventions for the programming of transactions;
- runtime control over the execution of transactions;
- features which minimize the need for serializable access by transactions and/or reduce the adverse consequences of rollback arising from deadlock.

The required support for these approaches is expected to be available in software-process programming languages and environments. While coding conventions may reduce the extent to which the flexibility of the FCM is realized, that flexibility is still available in situations in which deadlock is not expected to be a problem.

7 Related Work

This section presents related work in consistency and transaction models and compares and contrasts that work with FCM.

7.1 Balzer's "Accommodating Inconsistency"

Balzer [3] has defined a consistency model in which inconsistency can be accommodated in a natural way. In his model, consistency is judged with respect to predicates over data. Processes may (apparently) update these data in such a way as to violate a predicate, thus introducing inconsistency. However, data which violate a predicate are guarded, and any process which depends on those data is prevented from accessing them. The need to accommodate inconsistency seems to be a fundamental requirement for software object management, and Balzer's model is one of the few that addresses this explicitly. Balzer's model differ's from the model introduced here in two important respects. First, in Balzer's model inconsistency arises when data are changed to violate an existing constraint. In the model introduced here, data cannot be changed by a process to violate a predicate that is enforced on that process. Instead, inconsistency arises when a predicate is newly enforced on data that do not satisfy it or when a separate process on which the predicate is not enforced changes the data to bring about a violation for a process in which it is enforced. Second, in

Balzer's model, guards are associated with the violating data. In the model introduced here, the violating data are unmarked, and processes uses the consistency-management constructs to protect themselves from violations. Balzer's model focuses on inconsistency; he does not introduce a transaction model.

7.2 Moss's Nested Transactions

Moss [13] has defined a nested transaction model that is widely referenced (for example, [18,11,23,12]) and which has become, in effect, a standard model. This model was originally developed for distributed, concurrent, nested transactions. In this model a given transaction may nest possibly concurrent subtransactions. The nesting transaction is serializable with respect to other transactions with which it is concurrent. The nested transactions compete for serializable access within the nesting transaction. The results of nested transactions are committed temporarily and are subsequently accessible by other subtransactions within the nesting transaction. However, the results of subtransactions are not finally committed and made available outside of the nesting transaction unless and until the nesting transaction commits. Moss's model makes no special provisions for consistency beyond the assurance of serializability. The transactions in the model are conventional in that they combine serializability, consistency, and recovery.

The FCM model generalizes Moss's nested transaction model in two respects. First, FCM offers several consistency management statements that can be nested in various ways. Second, while FCM allows nested transactions in which the results of subtransactions are eventually committed and made visible (or aborted) depending on the nesting transaction, it also allows subtransactions to be separately committed and made visible (or aborted) independently of the nesting transaction. The rules governing serializability and recovery for nested transactions in FCM are presented in Section 4.4. The issue of deadlock in nested transactions is discussed in Section 6.3.

7.3 Abstraction

Moss, Griffeth, and Graham [14] address the issue of abstraction in recovery management. They define a hierarchical model of abstract actions in which actions at one level are implemented in terms of programs of actions at the next lower level. They further define abstract serializability and atomicity as properties of logs of abstract and concrete actions. They identify some relatively general conditions for correctness of serializability and atomicity in this model and suggest algorithms for serializability and recovery. Recovery management is important for consistency management in general and FCM in particular, but is beyond the scope of this paper. The relevant contribution of Moss, Griffeth, and Graham is to introduce a control model with abstraction. A limitation of their approach is that serializability and recovery depend on conditions that are not strictly abstract; they require access to operations at the concrete level.

Abstract data types are also introduced in POSTGRES [22,20]. In the POSTGRES data model, abstract types are defined in terms of their concrete implementations; thus abstraction is limited to a single level. Moreover, this concrete information is available to the recovery system.

Because abstraction is so important in software engineering (for example, [6,2,7]) the reliance on concrete information may be problematic for software process programming. An interesting issue for software process programming and FCM is the extent to which recovery management at the abstract level can be addressed without reliance on information about the concrete level.

7.4 Constraint Enforcement by Triggers

Some systems, for example HiPAC [12] and the software environment proposed in [21], provide for the enforcement of implicit constraints by triggers. This approach provides flexibility for consistency maintenance but puts the burden of maintenance on the programmer. A problem with this approach is that the constraints are not explicit. It can be difficult to extract a characterization of consistency from the triggers or to be assured that desired constraints will be enforced. Another problem is that inconsistency persists while the triggered operations are under way, so that it can be difficult to tell when a consistent state obtains.

AP5 [8] combines explicit constraints with triggered repair actions. The repair actions are invoked within a transaction if that transaction would violate specified constraints. However, constraint enforcement is not solely the responsibility of triggered actions. If a transaction, including repair actions, fails to satisfy applicable constraints, then the transaction is rolled back. A similar approach is taken in APPL/A [23], which combines triggers with the FCM consistency and transaction models. Constraints, i.e. enforced predicates, are stated explicitly. Triggered actions can be used within a transaction (e.g. suspend statement) to help assure the satisfaction of enforced predicates. However, any transaction that ultimately violates an enforced predicate is automatically rolled back.

7.5 Pu, Kaiser, and Hutchinson's Split- and Join-Transactions

Pu, Kaiser, and Hutchinson [18] offer an extended transaction model with operations to "split" and "join" transactions while preserving *commit* serializability. For convenience this model will be referred to as the SJT model. The SJT model is concerned strictly with transactions; it does not make any special provisions for consistency, and so it is not a consistency model in the sense that FCM is. Like the FCM transaction model (and unlike that in [11]) the SJT model is concerned with serializable access to a given version of an object and does not rely on persistent versions (as in [26]) to enhance concurrency. However, the SJT model is more flexible in certain respects than that of FCM.

The split- and join-transaction operations enable the database operations of transactions to be subdivided or combined for purposes of commitment while preserving serializability of the operations at the time of commitment. The split-transaction operation splits the current transaction into two new transactions which commit separately. The join-transaction operation "joins" a given transaction to the current transaction so that their results commit together. These operations preserve the serializability of the transactions which finally commit, hence the idea of commit serializability. However, because of splitting and joining, the transactions which finally commit may not correspond in a simple way to the transactions originally invoked.

The motivation behind split-transactions is to allow some of the results of a transaction to be committed and made available before the rest of the transaction terminates. The motivation behind join-transactions is to allow the results of separate transactions to be combined so that they can be committed and made available together. Thus the split-transaction operation increases access to data while the join-transaction operation reduces it.

An important characteristic of the SJT model is that transactions can be dynamically controlled by users. In particular, the split- and join-transaction operations can be invoked dynamically and need not be anticipated in advance. The FCM model assumes that the combined or separate commitment of a transaction is determined at the time the transaction is invoked. Thus FCM is less flexible with respect to transaction commitment than is the SJT model. (Some dynamic flexibility could be introduced into FCM by defining versions of the CM statements for which separate or combined commitment is determined conditionally at commit time. However, it remains to be seen whether this flexibility would be justified in light of the complexity it would add to the model. In particular, the need for this kind of flexibility may be reduced in a process-programming context by the proper design of programmed transactions, including highly interactive transactions.)

An effect somewhat like that of split-transactions (but static) can be achieved in FCM using separate composed transactions. The results of such a transaction can be released prior to (and independently of) other results of the including transaction. An effect somewhat like that of join-transactions (but again static) can also be achieved in FCM using combined composed transactions. In this case the results of an included transaction are held uncommitted as part of the including transaction (but are still available within the including transaction). In general it should be possible to encode any *given* split- or join-transaction schedule in FCM by working backward from the resulting split or joined transactions and constructing FCM combined or separate composed transactions which conform to the rules for data sharing and serializability defined in [18].

Apart from the issue of dynamic control of transactions, there is another important difference between the SJT and FCM transaction models. In the SJT model, the transactions resulting from a split-transaction operation may need to be serialized if there are data dependencies between the two (e.g. one of the new transactions reads data written by the other). In this case the second transaction in the serial order must be aborted if the first transaction is aborted. In contrast, with an FCM separate subtransaction, the results are committed altogether independently of the including transaction and may be committed even if the including transaction fails.

7.6 Korth and Speegle's Correctness without Serializability

Korth and Speegle [11] present a formal model of transactions which allows various relaxations of conventional serializability. Their motivation for relaxation of conventional serializability is to support long-duration "transactions" such as those required by design processes. Their model includes

- Nested transactions, based on Moss's model, but with a partial order on subtransactions.
- Predicates which define a database invariant and pre- and postconditions for transactions.
- Persistent versions.

For convenience, this model will be called the NTPPV model below.

The preconditions for a transaction describe the database state in which the transaction is assumed to execute correctly. The postconditions describe the database state that results from the correct execution of the transaction. For top-level transactions the postcondition must be consistent with the database invariant. For nested subtransactions the postcondition need not satisfy this invariant; it is assumed that violations of the invariant by one subtransaction will be corrected by others (and, if not, then presumably the whole top-level transaction will be undone).

Persistent versions of objects are used to enhance concurrency. Each update of an object creates a new version of it, and the old versions are retained. Transactions which require potentially conflicting access to an object can be given concurrent access to different versions of that object. However, in this model, each concurrent transaction on an object produces a different version of the object as a result. Korth and Speegle do not address the problem of the "merging" of these versions when a single result is desired from a concurrent process.

FCM, like the SJT but not NTPPV models, does not include any notion of persistent versions. Both NTPPV and FCM support nested transactions. The NTPPV model admits more classes of serializability while the FCM model is more flexible with respect to commitment in that it allow some of the results of some subtransactions to be committed even if the including transaction fails.

Both NTPPV and FCM have a view of consistency defined in terms of predicates. The NTPPV model has a conventional view of database consistency in that it assumes that the database is characterized by an invariant predicate, and it makes no provision for the evolution of this invariant over time. In contrast, FCM explicitly assumes that the database is not characterized by an invariant and that constraints on the database will indeed evolve over time. The NTPPV and FCM models are more similar with respect to the consistency of transactions. The NTPPV model applies predicates as pre- and postconditions to transactions; these are not, strictly speaking, enforced during a transaction, but subtransactions have their own pre- and postconditions which can assure that indicated constraints are enforced within a nesting transaction. FCM applies predicates as postconditions to individual operations and transactions. Some predicates may be suspended during a composite operation (e.g. during a suspend statement) and enforced only as postconditions to that composite operation. However, other predicates may continue to be enforced during the transaction as postconditions to each nested transaction or operation. Additionally, an enforce statement can be used to specifically impose constraints that would not otherwise be enforced in the subtransaction. Each of these models provides a uniform mechanism for expressing consistency throughout a hierarchy of nested transactions.

8 Summary

FCM is a flexible consistency model for software process programming languages; it includes a flexible transaction model. FCM is based on the premises that software processes require flexible control over when and where constraints are enforced, that software processes must be able to accommodate inconsistency, and that the programming of software processes can benefit from the use of relatively specialized but composable control constructs for serializability, atomicity (recovery), and constraint enforcement.

Consistency in FCM is defined with respect to predicates over data. Predicates may be enforced within a process, in which case they have the effect of postconditions on operations by that process on the data: an operation must leave the data in a state which satisfies all enforced predicates or the operation is rolled back. Thus enforced predicates act something like constraints. The scope of predicates may be global or local. A global predicate must be included in any program that accesses the data to which the predicate refers; a local predicate may optionally be included in any such program. Global predicates may be mandatorily enforced by default, while some global predicates and local predicates may be optionally enforced by default. The default enforcement of optionally-enforced predicates may be turned on and off dynamically.

FCM also includes five block statements with transaction-like properties:

- The **serial** statement, which provides simple serializable read and/or write access to data.

- The **atomic** statement, which provides serializable read and/or write access to data, with logging and recovery in the event of exception propagation.
- The **suspend** statement, which suspends the enforcement of designated predicates within its scope. It also provides serializable access to the data designated in those predicates, with logging and recovery in the event that the statement terminates in violation of an enforced predicate.
- The **enforce** statement, which imposes the enforcement of designated predicates within its scope. (This statement does not provide serializable access or recovery.)
- The **allow** statement, which is similar to the suspend statement, except that it allows the perpetuation of existing predicate violations.

Individually these statements are relatively specialized compared to conventional transactions. However, they can be composed to achieve the effects of conventional transactions and nested transactions as well as many other effects. The execution model on which the transaction model of FCM is based is specially designed for programming languages. It includes not only concurrency and nesting, but subroutines and abstract data types. It also provides "separate" nested transactions which can commit (or abort) independently of the enclosing transaction.

FCM accommodates inconsistency in a natural way. Inconsistency can arise when a predicate is newly enforced, or when one process causes data to be inconsistent with respect to a predicate that is enforced on another process. (However, note that no process can violate a predicate that is enforced on itself.) When inconsistency arises, it can be accommodated or repaired by turning off the enforcement of the predicate, by using a suspend statement to repair it, or by using an allow statement to accomplish some work even if the predicate is not repaired. Processes can protect themselves from inconsistency by controlling the enforcement of predicates directly, by using enforce and suspend statements, and by serializing access to data to preclude violations by other processes.

Taken together the features of FCM provide great flexibility in consistency management. They provide global and local control over when and where predicates are enforced, they support serializable access to data under a variety of conditions, and they allow inconsistency to arise and be handled in a natural way. The model is especially adapted to programming languages, and it includes constructs which facilitate the programming of consistency management. FCM should thus greatly enhance consistency management for software process programs and the resulting software processes.

9 Acknowledgements

The design of APPL/A has been supervised by Lee Osterweil and Dennis Heimbigner. With respect to the present work the author wishes especially to thank Frank Belz for his reading of the first draft; Frank, Christine Shu, Lolo Penedo, and Sandy Schreyer at TRW comments on presentations of later material; and Dennis Heimbigner for his reading of a later draft and discussion of many of the ideas included here. Thanks also to other members of the Arcadia consortium who have provided comments on this material and on related work in APPL/A.

References

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

[2] Russell J. Abbott. *An Integrated Approach to Software Development.* John Wiley & Sons, New York, 1986.

[3] Robert Balzer. Tolerating inconsistency. In *Proc. 5th International Software Process Workshop*, October 1989. Kennebunkport, Maine.

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

[5] Philip A. Bernstein. Database system support for software engineering – an extended abstract. In *Ninth International Conference on Software Engineering*, pages 166–178, ACM, 1987.

[6] Grady Booch. Object-oriented development. *IEEE Trans. on Software Engineering*, SE-12(2):211 - 221, February 1986.

[7] Lori A. Clarke, Jack C. Wileden, and Alexander A. Wolf. *Graphite: A Meta-Tool for Ada Environment Development.* Technical Report COINS TR 85-44, University of Massachusetts at Amherst, Computer and Information Science Department, Software Development Laboratory, Amherst, Massachusetts 01003, November 1985. Working draft.

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

[9] Klaus R. Dittrich, Willi Gotthard, and Peter C. Lockemann. DAMOKLES - a database system for software engineering environments. In *International Workshop on Advanced Programming Environments*, IFIP WG2.4, 1986.

[10] 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.

[11] Henry F. Korth and Gregory F. Speegle. Formal model of correctness without serializability. In *Proc. of the ACM SIGMOD International Conference on the Management of Data*, pages 379 – 386, 1988.

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

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

[14] J. Eliot B. Moss, Nancy D. Griffeth, and Marc H. Graham. Abstraction in recovery management. In *Proceedings ACM SIGMOD '86 International Conf. on Management of Data*, pages 72 – 83, 1986.

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

[16] Maria H. Penedo, Erhard Ploedereder, and Ian Thomas. Object management issues for software engineering environments – workshop report. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 226 – 234, ACM, 1988.

[17] Maria H. Penedo and William E. Riddle. Guest editors' introduction: software engineering environment architectures. *IEEE Trans. on Software Engineering*, 14(6):689–696, June 1988.

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

[19] Jayashree Ramanathan and Soumitra Sarkar. Customized assistance for software lifecycle approaches. *IEEE Trans. on Software Engineering*, 14(6):749–757, June 1988.

[20] 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.

[21] Soumitra Sarkar. *Data Model and Persistent Programming Language Features for Integrated Project Support Environments.* Technical Report OSU-CISRC-6/89-TR22, Computer and Information Science Research Center, The Ohio State University, Columbus, Ohio, 43201, June 1989.

[22] 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.

[23] 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, Univ. of Colorado, Dept. of Computer Science, Boulder, Colorado 80309, October 1989.

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

[25] Jeffrey D. Ullman. *Principles of Database Systems. Computer Software Engineering Series*, Computer Science Press, Rockville, Maryland, second edition, 1982.

[26] J. Walpole, G. S. Blair, J. Malik, and J. R. Nicol. A unifying model for consistent distributed software development environments. In *Proc. Third ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 183 – 190, November 1988. Special issue of SIGPLAN Notices, 24(2), February, 1989.