# Consistency Management for Complex Applicationss

Peri Tarr⋆
Lori A. Clarke †

CMPSCI Technical Report 97–27
January 1997

⋆IBM T.J. Watson Research Center
30 Saw Mill River Road
Hawthorne, NY 10532

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

---

† This work was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract F30602-94-C-0137.

# Consistency Management for Complex Applications*

Peri Tarr
IBM T.J. Watson Research Center
30 Saw Mill River Road
Hawthorne, NY 10532
tarr@watson.ibm.com

Lori A. Clarke
Department of Computer Science
University of Massachusetts at Amherst
Amherst, MA 01003
clarke@cs.umass.edu

## Abstract

Consistency management is an important requirement in many complex applications, but current programming languages and database systems provide very limited support for it. To address this limitation, we have defined a consistency management model, based on what we perceive to be the underlying requirements of these applications. This paper presents a motivating example that illustrates some typical consistency management requirements, discusses the requirements in terms of both functionality and cross-cutting concerns that affect how this functionality is provided, and then proposes a model of consistency management. This model has been implemented in the PLEIADES object management system, and we describe some design and implementation issues that arose in instantiating the model. Although the current implementation has some drawbacks, users have been extremely pleased with the system and have provided us with valuable feedback that has influenced the model and our future research plans.

***Keywords:*** *Consistency management, inconsistency management, object management, software engineering environments*

* This work was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract F30602-94-C-0137.

# 1. Introduction

One of the difficult tasks that arises in many complex applications is the need to define and maintain *consistency* among objects. One or more objects are said to be consistent if they are in states that satisfy some condition(s) for acceptability or correctness. An example of a complex application needing support for consistency management is a software engineering environment (SEE). In this domain, for instance, a source code module may be said to be consistent if it compiles, while the source code module may be said to be consistent with respect to its object code if the object code's time stamp is later than that of the source code. SEEs must facilitate the specification and enforcement of consistency definitions over objects to enable such activities as automated verification of task completion and detection of (potentially) erroneous manipulations of artifacts of the software engineering process. Although many applications and domains can benefit from consistency management, we draw our motivation and examples for this paper from SEEs, the domain with which we are most familiar.

*Consistency management* is the process of controlling the manipulation of objects to ensure that their consistency definitions are respected. Consistency management comprises the definition of consistency conditions, identification of consistency violations, reestablishment of consistency following violations, and control over access to objects that do not satisfy their consistency conditions. Consider, for example, a source code module that has "is compilable" as a consistency condition. This condition could be violated upon any modification to the source code. Depending on the phase of development, a project manager may or may not want to allow the source code to be in an inconsistent state. If the violation is acceptable, as is often the case, then the change might be allowed, but some kinds of manipulations of the inconsistent object might be precluded (e.g., it could not be released or tested). If the violation is unacceptable, a consistency management mechanism might reject the change and roll the module back to its previous, consistent state, thus preserving consistency.

Managing object consistency is an important, but difficult, task, for a number of reasons. One is the degree of diversity of object and consistency semantics that must be supported. Different kinds of objects may require different consistency definitions and enforcement semantics. For example, some kinds of objects require *invariant* consistency definitions (i.e., the consistency definition may not be violated and any activity that threatens to violate the definition must be precluded), while other objects will allow temporary violations of their consistency definitions with the expectation that the violations can be repaired, either immediately or eventually. Second, the set of consistency definitions and enforcement semantics that apply to any given object may change during the lifetime of the object. For example, during development phases, "is compilable" may not be applicable to a source module, but once the project reaches a release phase, this consistency definition might have to be enforced. The broad spectrum of object, consistency definition, and consistency enforcement semantics results in some fairly challenging requirements on consistency management systems. While some existing systems include support for consistency management, we are unfamiliar with any system that supports the wide range of consistency semantics that we have found to be needed in advanced applications, such as SEEs.

This paper examines a number of issues involved in supporting consistency management. Section 2 provides a small but typical example of an advanced application to illustrate some of the consistency management needs that such applications have, and then uses the example to help motivate a set of requirements on consistency management systems. Section 3 describes a model of consistency management that satisfies those requirements. We have implemented much of

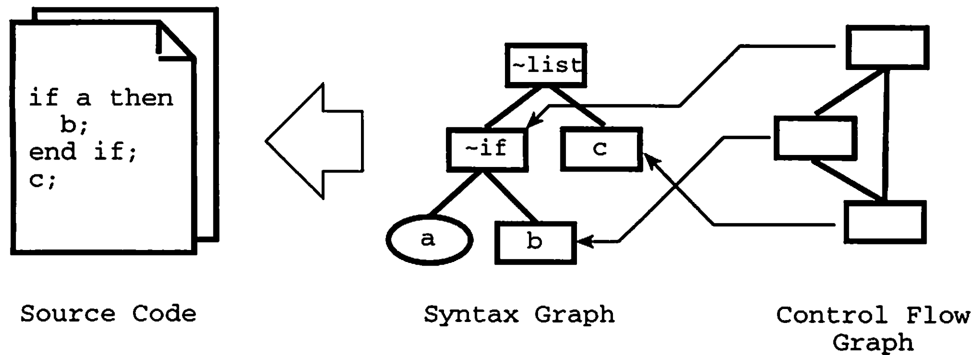


Figure 1: An SEE Motivating Example.

this model in the PLEIADES object management system [42], and in Section 4, we discuss various tradeoffs involved in satisfying the requirements in PLEIADES, both at the design and implementation levels. PLEIADES has been used in the implementation of several software engineering applications. Section 5 describes some of the uses of PLEIADES' consistency management system and uses client feedback to evaluate the requirements on, and model of, consistency management presented in Sections 2 and 3. In Section 6, we examine some related research in the area of consistency management. Finally, Section 7 discusses ongoing and future work.

## 2. Motivating Example and Requirements for Consistency Management

To illustrate some typical consistency management needs in complex applications, we use an example from the Arcadia SEE project [20]. In this example, there are three types of objects: source code, abstract syntax trees (ASTs), and control flow graphs (CFGs), along with some corresponding consistency definitions. As depicted in Figure 1, each AST node is associated with the source code from which the AST was created, and each node in a CFG is connected to with the root of the AST subgraph that elaborates the statement associated with the CFG node. These three kinds of objects may be subject to several complex consistency definitions, including:

- Acyclic: ASTs are, by definition, trees. Thus, they may not include cycles or shared substructure.

- Up-to-datedness: To ensure that applications only manipulate ASTs and CFGs that correspond to the current state of the associated source code, an up-to-datedness constraint is enforced among these three abstract data types. This constraint indicates that a set of interrelated ASTs, CFGs, and source modules are up-to-date with respect to each other either if the time stamp on the source is less than the time stamp of the AST, which, in turn, is less than the time stamp on the CFG, or if a manager agrees that the objects are mutually consistent.

- No def/use errors: Def/use errors can be found via static analysis, and they include such anomalies as a reference to a variable that is not defined. During the later stages of

program development, CFGs may be subject to a constraint that they are consistent only if they do not contain any def/use errors.

These three consistency conditions are quite different from each other in nature. The acyclic property represents an invariant on the AST abstract data type, and, as such, it may not be violated under any circumstances—any attempt to introduce a cycle or shared substructure can be viewed as an error and should be prevented. The up-to-datedness constraint, on the other hand, is one that is expected to be violated during the normal evolution of code. It will be violated, for example, any time a developer modifies part of a program (either by editing the source code or by changing a visual depiction of the AST or CFG). Such violations are neither abnormal nor erroneous, and they should be permitted. It is expected, however, that once a violation has occurred, consistency can be reestablished by identifying the scope of the change and recomputing the corresponding parts of the source code, AST, or CFG (depending on which structure was modified). A failure to reestablish consistency, however, might be considered an abnormal condition that might or might not be permissible, depending on the stage of development. "No def/use errors in CFGs" is yet another kind of constraint. Like the up-to-datedness constraint, "no def/use errors" is a consistency condition that is expected to be violated during the normal evolution of a program. Unlike the up-to-datedness constraint, however, it may be possible to violate "no def/use errors" during some stages of development and for the violation to be acceptable, at least for some period of time. While the CFG is in an inconsistent state, however, a different set of operations might be permitted on the CFG, and on some of its related objects, than are permitted while the CFG is in a consistent state. For example, it should not be possible to invoke the **release_system** operation on a source module object whose corresponding CFG fails to satisfy the "no def/use errors" condition, but it may be possible to use a def/use error visualization tool on the inconsistent CFG. Thus, when consistency violations cannot be repaired immediately, it may be necessary to *tolerate* the inconsistency for some period of time [4, 36, 39] and to manage access to the inconsistent objects appropriately.

These fairly simple examples illustrate some of the kinds of functionality that are needed to facilitate the definition and management of object consistency. They also demonstrate some "cross-cutting" requirements, which are constraints on the ways in which the consistency management functionalities are provided. These functional and cross-cutting requirements are discussed below.

## 2.1 Functional Requirements

As the above example indicates, there are many aspects of consistency management. After carefully examining the needs of several complex applications, we believe that a consistency management system must provide users with the ability to do the following:

Define consistency conditions: Consistency management starts with the definition of what it means for one or more objects to be consistent. Essentially, this definition partitions the space of possible object states. A simple and very common partitioning is "consistent" and "inconsistent," but partitionings may be more complex. A consistency management system should facilitate the specification of any kind of partitioning.

Determine when to detect violations: As shown by the examples above, different kinds of objects may have consistency definitions that require detection of violations at different points. For example, the acyclic invariant on ASTs may not be violated, which means that potential

violations must be detected before they actually occur; the up-to-datedness constraint can be violated, but the violation must be detected immediately so that it can be repaired; and violation of the "no def/use errors" consistency definition can occur lazily—it need only be detected prior to the use of any operation that might not be allowed while the CFG is in an inconsistent state. Thus, it must be possible to detect (potential) violations of consistency conditions in different ways, as needed to enforce the required kinds of consistency semantics.

**Specify enforcement semantics**: It must be possible to define appropriate responses to (potential) consistency violations. As seen in the examples, responses to consistency violations may range from outright rejection of the action that caused the violation, to rolling the affected objects forward into a new, consistent state, to allowing the affected objects to remain inconsistent and controlling access to them. In addition, enforcement semantics must consider that the initial repair action may not be successful and that subsequent actions may have to be considered.

**Manage inconsistency**: Even with a rich set of repair actions, it may not always be possible, or even desirable, to return an object to a consistent state. When inconsistency is to be tolerated, however, it is sometimes necessary to restrict access to the inconsistent objects. In the example, the CFG could be graphically depicted when the "no def/use errors" condition is violated, but the source code associated with that CFG cannot be released.

**Dynamically change the consistency information associated with an object**: The consistency constraints that apply to a given object need to change over time. For example, the "no def/use errors" and up-to-datedness constraints can both apply to CFGs, but the former typically is enforced only during release phases, and not during development phases, while the latter might be enforced during a CFG's entire lifetime. Similarly, the repair action associated with the up-to-datedness constraint may change; during development, recompilation might be invoked automatically, but during the release phase, a manager might be asked to approve the change.

### 2.2 Cross-Cutting Requirements

The functionality described above can be implemented using capabilities found in most modern programming languages and some database systems, but not without extensive programming. One of our goals in this work was to describe a model of consistency management that provides more powerful building blocks than those currently found in programming languages and databases—primitive capabilities that are easy to use, but general enough to permit the definition of a broad range of consistency management semantics. Towards satisfying this goal, we have further constrained the set of functionalities discussed above with a set of cross-cutting requirements [42] that describe more specifically *how* these capabilities should be provided to produce a flexible, easy-to-use consistency management system.

**Completeness**: *Computational* completeness supports the definition of arbitrarily complex algorithms, both for determining whether or not objects satisfy consistency conditions and for specifying enforcement semantics. *Type* completeness provides the ability to associate consistency conditions and enforcement mechanisms with any type of object.

**Meta-data**: To make decisions dynamically, applications require information about their run-time state or environment, which is commonly referred to as meta-data. Information about the

set of consistency conditions that are currently enforced on an object and about an object's consistency status are examples of the kinds of meta-data that may be required.

**Generality/heterogeneity**: Previous research (e.g., [43, 20, 36, 7]) has demonstrated that different programming paradigms and models are appropriate for different kinds of applications. *Generality* means that a consistency management system must provide a set of primitive capabilities that facilitate the implementation of alternative consistency management paradigms. *Heterogeneity* means that a consistency management system must allow alternative consistency management models and implementations to coexist peacefully and, when appropriate, to be used together in an integrated manner.

**First-class status and identity**: First-class status provides the ability to treat all objects uniformly. The ability to pass a consistency condition or action as a parameter to an operation is an example of this requirement. Identity means that a given entity has a unique identifier that is separate from its state. First-class status and identity facilitate the definition of consistency constraints among constraints and/or actions to facilitate, for example, the decomposition of constraints and the representation of preconditions on conditions (such as checking the up-to-datedness condition before the "no def/use errors" condition, since a failure to satisfy the former automatically results in the failure of the latter). Identity facilitates sharing, which makes it easier, for example, for enforcement semantics to be shared among constraints. First-class status and identity also facilitate reflection [13].

## 3. Model of Consistency Management

Based on the requirements described in Section 2, we have defined a model of consistency management, which is presented here. A formal specification of this model is given in [41]. In addition, we have incorporated much of the model into the PLEIADES object management system [42, 41] in order to evaluate the model and to better understand implementation considerations. This section describes the consistency management model.

To facilitate the description of the consistency management model, we assume an abstract data type (ADT) programming model. We view all objects as instances of ADTs, which means that access occurs solely via *operation invocation*. This assumption implies that the only way to violate a consistency condition is by invoking an operation on an object.[1] We rely on this assumption throughout the remainder of this section.

The consistency management model recognizes and enforces a set of semantics specified in the form of *consistency constraints*. A consistency constraint comprises a *consistency condition*, a set of points at which violations are to be detected, *enforcement semantics*, and *inconsistency management semantics*. An *instantiation* of a consistency constraint represents the enforcement of a given constraint on one or more objects. Both constraints and instantiations can be changed throughout execution, providing applications with extensive dynamic control over consistency management. We describe conditions, violation detection, enforcement semantics, and inconsistency management below.

[1] Observe that temporal constraints can be modeled using ADTs as well—i.e., the clock is an instance of an ADT on which operation to change time are invoked. Note that we do not propose this as an *implementation* mechanism, but rather, as a *modeling* mechanism.

### 3.1 *Consistency* Conditions

The specification of what it means for objects to be consistent is accomplished by defining *conditions*. A condition is a function whose return value is a *consistency status* flag that indicates in what state the object is. By default, the consistency status values are in the set {**consistent**, **inconsistent**, **partial**, **unknown**}. **Consistent** and **inconsistent** mean that the condition does or does not hold, respectively. It is possible to describe decomposition relationships among conditions (e.g., to facilitate incremental condition satisfaction). **Partial** means that some, but not all, subcomponents of a condition evaluate to **consistent**. **Unknown** means that not enough information is available to the status of one or more objects [37]. This may happen, for example, if human intervention is required, but not available, to determine whether or not a condition is satisfied, or if concurrency control conflicts arise that preclude access to objects whose states affect the status of one or more objects with respect to a condition. **Consistent** and **inconsistent** are the most commonly used partitions, though the other values may also be needed in some circumstances (e.g., in software process programming). The set of consistency status values must be user-extensible, to facilitate the definition of application-specific consistency status values.

Conditions are computationally complete, which means that any necessary condition can be specified. They may be enforced on objects statically or dynamically, on a per-instance or per-type basis. Applications can check, at any time, to see what the consistency status of one or more objects is, with respect to a given condition.

### 3.2 Violation Detection

Since we employ an ADT model, it is only possible to modify or examine the state of an object, and thus, to violate an enforced constraint or view an object in an inconsistent state, by invoking an operation on an object. Thus, information about when to detect violations is specified in terms of a set of tuples of the form <*operation, when*>, where *operation* is the name of an operation in which a condition should be checked, and *when* is in the set {**preinvoke**, **precondition**, **postcondition**, **postinvoke**}. **Preinvoke** means that a condition will be checked prior to the invocation of the specified operation. This is particularly useful in cases where failure to satisfy the condition precludes the invocation of the operation. **Preconditions** are checked during the execution of the specified operation, but before the operation takes any other actions, while **postconditions** are checked after the operation has performed its task, but before it terminates. Pre- and post-condition checks are used in cases where the runtime context in which the operation executes is important to the checking of the condition, and for cases in which the operation may have to be prevented from committing due to a violation. **Postinvoke** means that a condition will be checked after the specified operation finishes executing and commits. In general, we believe that postcondition checks are more common than postinvoke checks, since postconditions can affect the commit of the operation while postinvoke checks cannot, but a postinvoke check may be useful, for example, in circumstances where the satisfaction of a condition depends on whether or not the specified operation actually committed.

The description of when (potential) violations should be identified can be done dynamically or statically, at the per-constraint, per-object, and per-type levels.

### 3.3 Definition of Enforcement Mechanisms

It is possible to define an *action* to be taken when the consistency of one or more objects is violated (with respect to a given condition). By default, the violation of a condition is assumed to be undesirable, so a diagnostic exception is raised. Actions are essentially procedures and are computationally complete, so any required action may occur in response to a violation. In general, actions are intended to prevent or correct a consistency violation, though they may perform any tasks deemed necessary, such as sending mail to a developer, logging the violation, etc.

Actions may be associated with selected conditions enforced on particular objects, and mechanisms are provided to specify this association both statically and dynamically. In addition, both instance- and type-level control are provided; thus, two objects of the same type could take different actions upon violation of the same condition.

Ideally, once its associated action has been run, the violated condition will be satisfied. Clearly, this need not be the case, however. For situations in which additional actions must be taken if the original fails to restore consistency, developers may specify *action chains*. Action chains are essentially postconditions on actions. They indicate what new action to take if a given action fails to restore consistency. These chains may be as long as needed and may be modified dynamically.

### 3.4 Inconsistency Management

It is possible that, after applying all actions in an action chain, a condition still will not be satisfied. Developers are, therefore, given the option of describing which operations on the object are permissible (or not permissible) while a given condition is in a state other than **consistent**. By default, objects are assumed not to be allowed to end up in any state but **consistent**; thus, if an action chain fails to restore consistency, a diagnostic exception is raised. Inconsistency management semantics can be associated with a given instance or type, and they can be changed dynamically. Different inconsistency management semantics also can be associated with different consistency status values.

### 3.5 Using the Model

To demonstrate how this consistency management model could be used, we now revisit the motivating example presented in Section 2 and describe how one of the three consistency definitions presented in that section, namely, the acyclic constraint, could be represented using the model. We employ PLEIADES-like syntax throughout this section to illustrate the concepts.

The acyclic invariant is modeled as a condition that is checked as a precondition to each insertion into, and edge redefinition of, an AST.[2] If the proposed insertion or edge modification would create a cycle or shared substructure, the update is prevented and a diagnostic exception is raised.

```
condition Is_Acyclic ( The_AST : AST;
                       Source_Node_For_New_Edge : AST_Node;
```

[2] Note that removal of nodes from an AST cannot cause cycles to occur, so this constraint need not be checked upon node removal.

```
                              Target_Node_For_New_Edge : AST_Node ) is
begin
   -- If the target node of the new edge does not already have a
   -- parent, the change is acceptable.  If it already has a
   -- parent, the change will introduce a cycle or shared
   -- substructure.
   if (Get_Parent (Get_Target (Target_Node_For_New_Edge))
       /= Null_AST_Node) then
      return Inconsistent;
   else
      return Consistent;
   end if;
end condition;

action Reject_Update ( The_AST : AST;
                              Target_Node_For_New_Edge : AST_Node ) is
begin
   -- First, report the error:
   Put_Line ("Attempted to define edge that introduced a cycle");
   -- Raise a diagnostic exception.  This will terminate execution
   -- of the update operation.
   raise Attempt_To_Violate_Acyclicity;
end action;
```

By default, **Is_Acyclic** is not enforced on instances of type AST, which means that the "treeness" invariant can be violated. The consistency management model provides both static and dynamic mechanisms to enforce **Is_Acyclic** on ASTs. The static mechanism is a declarative statement which indicates that, at least initially, **Is_Acyclic** should be enforced on instances of type AST:

```
check Is_Acyclic in Set_Edge as precondition;
```

The dynamic mechanism is provided in the form of two operations, **Enforce_Constraint** and **Relax_Constraint**, which control the enforcement of constraints on particular instances of a type:

```
procedure Enforce_Constraint
   ( The_Condition           : Condition_Name;
     On_Object               : Object_Type;
     Enforcement_Mechanism   : Enforcement_Info );
procedure Relax_Constraint
   ( The_Condition           : Condition_Name;
     On_Object               : Object_Type;
     Enforcement_Points      : Enforcement_Info );
```

Thus, applications may enforce or relax constraints on objects at any point during the objects' lifetimes. Although dynamic control is not required for the enforcement of invariants, like **Is_Acyclic**, it would be very useful for constraints, like "no def/use errors," that apply during more limited periods of time during an object's lifetime.

## 4. Design and Implementation Concerns

The model of consistency management presented in the previous section is both general-purpose and language-independent. To enable developers to use it, the model must be instantiated for, and bound into, a particular programming language.[3] Developers can then continue to use their

[3] A programming language enhanced with capabilities like consistency management, persistence, concurrency control, and other object management capabilities is typically referred to as a *database programming language* [3].

favorite language for software development and draw on the consistency management extensions. PLEIADES represents one such instantiation of the model, for the Ada programming language [44].

Instantiating the consistency management model required us to address a number of design and implementation issues; indeed, the model could have been instantiated in any number of ways, depending on which decisions we made. While some of these are specific to an Ada instantiation, many are general issues that must be addressed by any instantiation. This section describes these issues and discusses justifications for, and the implications of, some of the decisions we made in implementing PLEIADES. A more detailed discussion can be found in [41].

**General Issues**: The purpose of imposing the cross-cutting requirements was to ensure the definition of a consistency model that is powerful and flexible enough to facilitate the description of many different consistency management semantics. With this flexibility comes a number of tradeoffs, however.

The requirement for computational completeness means that any necessary consistency specification can be defined. The negative side of computational completeness is the difficulty of reasoning about a computationally complete formalism. The ability to reason about consistency specifications and instantiations is, however, very important. It can produce, for example, information about conflicting or redundant consistency conditions, and about the set of operations that could violate a given constraint. On the other hand, formalisms that are more amenable to analysis and reasoning are not complete, so they restrict the set of possible consistency conditions. Any instantiation of the consistency model described in Section 3 must decide where it falls on the spectrum between completeness and analyzable.

As noted earlier, first-class status and identity of objects provides the ability to model relationships among, and constraints over, any kinds of objects. The identity requirement, however, can lead to a fairly serious problem in implementing a consistency management system. The problem, which we call the *container problem* [41], arises when the consistency status of one object depends on the states of other, independent objects. Perhaps the best-known example of the container problem is the dangling references problem. For example, an application might destroy a node in an AST without realizing that other nodes still refer to it. The destroyed node affects the consistency of those that refer to it, according to a referential integrity constraint. The container problem is pervasive in many software systems and is particularly problematic in its effect on consistency management. Numerous ad-hoc solutions to this problem have been used, including domain-specific approaches like garbage collection (to address the dangling reference problem) and general-purpose approaches like invertible pointers, wrappers, polling, and event-based notification, but no existing approach scales to address all forms of the container problem in the context of consistency management. We are developing an approach to address the container problem by identifying different kinds of container problems and different object features that affect the selection of the most appropriate approach for managing consistency in given contexts.

The dynamic control requirement provides a great deal of flexibility. Satisfying this requirement raises some important issues, however. One is the inverse relationship between dynamic control and optimizability and analyzability—more dynamic control implies fewer opportunities for optimization and analysis. This may be acceptable in many situations, but in cases where a developer knows declaratively that they do not require dynamic control (e.g., in the case of

enforcing invariants, like "is acyclic"), it might be desirable to include mechanisms by which this information could be imparted to the compiler. The increased potential for optimization and analysis comes at the cost of additional complexity, however; thus, the selection of a point on the optimizability vs. dynamic control spectrum must occur as part of the mapping of the consistency model to a particular programming language.

**Current status**: The current implementation of PLEIADES supports much of the model described in Section 3 and addresses many of the functional and cross-cutting requirements described in Sections 2.1 and 2.2, respectively.

PLEIADES is implemented as a preprocessor for Ada. Developers describe abstract data types using primitives PLEIADES provides, as illustrated in Section 3, and PLEIADES produces an Ada package, called an *interface package*, which provides a set of type and operation definitions for creating, manipulating, and enforcing consistency over instances of those abstract data types. Applications can then use these packages as they would use any other.

We selected a preprocessor implementation strategy because it was the most expedient way to develop a prototype for evaluation, especially since we did not have access to an open Ada compiler and because our potential users felt more comfortable with an extension that created standard Ada code, rather than becoming dependent on a one-of-a-kind compiler. The selection of a preprocessor strategy had some negative consequences, however. In the area of consistency management, the primary one is that developers are restricted in the set of instantiations of constraints they can describe declaratively. Specifically, since inputs to PLEIADES describe *types*, not *instances*, only declarative instantiations of type-level constraints can occur. Instance-level instantiations must occur dynamically. In addition, PLEIADES cannot perform any analyses that involve the client code, such as identifying consistency conditions that are defined but never used, or statically determining when an attempt is made to enforce a consistency condition on an object to which it does not apply.

PLEIADES supports all aspects of the consistency model presented in Section 3, with a few exceptions. First, the set of consistency status values that can be returned from conditions is predefined to be **consistent** and **inconsistent**, and this set is not currently extensible. Second, PLEIADES can perform consistency condition checking as **preconditions** and/or **postconditions**, but it does not yet implement support for **preinvoke** and **postinvoke** checks. Third, action chains are not supported adequately in the current version of PLEIADES. Specifically, if developers wish to define an action chain, they must define each action so that it invokes the next action in the chain. Fourth, PLEIADES does not provide an adequate degree of support for inconsistency management. In particular, it does not provide a simple, declarative means of indicating which ADT operations can or cannot be invoked while a given object is inconsistent. For most of the current limitations, we believe that users can achieve the desired semantics using the existing capabilities, but this requires more programming intervention than we believe is desirable and permits fewer opportunities for analysis and automated support. These restrictions exist because we did not initially recognize the need for these capabilities, but user feedback indicated that they would be useful. None are particularly problematic to implement, and we plan to include them in future versions of the system. Finally, PLEIADES does not, at present, satisfy two of the cross-cutting requirements: it does not make conditions and actions first-class entities, and it is not type-complete—constraints can be enforced only on a subset of types. The former restriction comes directly from Ada, which does not satisfy the first-class status requirement, and is discussed in Section 5. The latter is a result of using a preprocessor implementation approach,

since we simply did not have the resources available to analyze all Ada types to the degree required to permit consistency management.

## 5. Experimental Evaluation

PLEIADES is currently in use in a number of real-world applications, both academic and industrial. It is, of course, difficult to quantify, and thus evaluate, functionality. In this section, we summarize feedback we obtained from PLEIADES users to help evaluate the PLEIADES prototype and the consistency model.

The evaluation we performed was based on information and feedback obtained directly from several PLEIADES users [1, 17, 28, 30, 38, 48]. The client applications about which we obtained information were a reusable components library [48], the Arcadia language processing tool set [46, 47], TAOS (Testing with Analysis and Oracle Support) [33], the Booch Object-Oriented Design process program (BOOD) [40], FLAVERS (Flow Analysis and VERification System) [14], an agenda management system [28], and an avionics validation and verification system [25]. The process we used to perform the evaluation was as follows. We constructed a questionnaire that included approximately fifty questions. The questions attempted to determine whether, and how, each PLEIADES client had used capabilities resulting from each of the functional and cross-cutting requirements, how closely the functionality provided satisfied the user's needs, and whether current limitations or existing features of PLEIADES caused the user difficulties. We then performed an evaluation of each user's experiences, based on the information provided. Once this evaluation was written, it was sent to the user for correction and feedback. A description of the complete evaluation we performed is outside the scope of this paper but appears in [41].

Support for consistency management was added to PLEIADES fairly late. Thus, by the time consistency management was implemented, several clients were already using the system and had managed to work around the lack of consistency management functionality. Some users reengineered their systems to use the new consistency management capabilities, but several did not. Thus, the amount of feedback we have about client use of consistency management is more limited than about other PLEIADES functionality.

The results of the evaluation suggest that, in general, the consistency management requirements and model we proposed are sound. Clients liked and made use of most of the capabilities associated with satisfying the functional requirements, and they made use of all the capabilities associated with the cross-cutting requirements. It was typically the case that problems reported were due to a failure to satisfy either a functional or cross-cutting requirement.

As noted in the previous section, feedback from users led to some changes in the consistency management model. The feedback we obtained pointed up some other noteworthy items as well. These include:

- PLEIADES' constraint enforcement mechanism was found to be too fine-grained for some kinds of objects. In particular, the number and complexity of constraints on the BOOD artifacts makes the cost of constraint checking very high; BOOD cannot tolerate the performance cost of checking these constraints upon each potential violation. Consequently, the BOOD artifact constraints are left unenforced much of the time, and they are checked manually by BOOD at appropriate times. This suggests a need to

associate constraint enforcement with *blocks* of operations, as well as with individual operations. This is similar to the transaction model used in database systems.

- One user noted that it was somewhat difficult to specify some kinds of inter-object constraints in PLEIADES. Specifically, because Ada operations are not first-class entities, constraints and actions in PLEIADES, which are modeled as operations, are not first-class entities. This limitation, combined with Ada's (and consequently, PLEIADES') static type model, means that only those constraints specified as part of an ADT's type definition can apply to instances of that ADT. To work around this limitation, this user had to define all of the ADTs to which inter-object constraints applied in the same specification, rather than separating them appropriately, which reduced the modularity of his application.

- In evaluating client use of PLEIADES, we have noted the pervasiveness of several general classes of constraints. These include:

  - **Up-to-datedness constraints**: Up-to-datedness constraints are used commonly to assure percolation of changes among related objects. They are often used to maintain "is derived from" relationships among objects, and they have been used to help identify the impact of a change, even in cases where repair was undesirable or not possible.

    Up-to-datedness constraints tend to be subject to roll-forward enforcement semantics—that is, violation is expected to occur during the normal evolution of objects, and it is often expected that repair can occur. Applications differ widely in the semantics they attach to a failure to repair violations of such constraints, however. Some rely on the success of the repair, to the extent that they cannot continue if repair fails. Others recognize that repair may not be successful and are prepared to try alternative repair mechanisms or to continue operating with inconsistent objects.

  - **"Well-formedness" constraints**: These constraints are used frequently to impose type and instance semantics that are not expressible using specification mechanisms present in standard programming language type models. For example, the "is acyclic" constraint in an AST is a well-formedness constraint.

  - **Operation constraints**: Many kinds of full or partial order relationships among invokable entities exist. For example, no **Pop** operation may occur on a stack object until the first **Push** is invoked. In most languages, these kinds of ordering constraints must be enforced manually, by checks included in operation implementations. This means that it is much more difficult to reason about, and change the enforcement of, such constraints. Among the kinds of operation constraints we found in PLEIADES clients were ordering of operations, condition checks, and action invocations.

  The ubiquity of these classes of constraints suggests that it would be beneficial to facilitate their description explicitly. Further work is needed to determine how best to leverage information about these classes of constraints to provide the most support for developers.

## 6. Related Work

Traditional programming languages are very limited in the ways they support consistency control. Strongly typed programming languages incorporate predefined notions of consistency in terms of conformance to type definition, but the set of violations that can be detected are usually restricted to criteria such as bounds checking and erroneous type usage; they do not support complex consistency definitions (e.g., well-formedness, up-to-datedness, etc.). Assertion (e.g., [34, 26]) and exception handling mechanisms (as in Ada [44] and CLU [23]) are specialized consistency management mechanisms that have been associated with some programming languages. Assertions are intended to describe invariant conditions of a running program and to specify actions to be taken upon detecting a violation of an invariant. Assertions are often used as an aid for debugging. Exceptions are intended to reflect unusual conditions and to specify actions to be undertaken if one of these conditions should arise. Exceptions are often used to support error processing. Assertion and exception handling mechanisms usually do not satisfy the cross cutting requirements for dynamic control over enforcement[4] or first-class status or identity of the conditions or actions associated with these mechanisms.

Many relational database systems support constraints. Their constraint enforcement mechanisms do not, however, satisfy most of the cross-cutting requirements. Relational databases do not support application control over constraint enforcement or invocation of different actions at different times—constraints are enforced at all times except during a transaction, when all constraints are relaxed. Relational databases support only roll-back semantics—if constraints are not satisfied at the end of a transaction, the effects of the transaction are undone.

Many database programming languages and object-oriented databases support only a limited, predefined set of consistency definitions, such as referential integrity (e.g., [27]) or programming language kinds of consistency definitions (e.g., [45, 2]), or they support consistency definitions over only a subset of types (typically collection types; e.g., [39, 11]).

Much research has been done in consistency management in the area of software process languages [31]. Some of these languages provide little or no support for product consistency management (e.g., [15, 9, 21]) or rely on consistency management support from an associated object management system (e.g., SLANG [5, 6]). Many software process languages incorporate conditions on product state as process control conditions, rather than as product consistency conditions. For example, EPOS [12] uses guards and postconditions to affect flow of process control; Grapple [18] and Interact [32] model conditions as goals; Melmac [16] and SLANG use conditional branching; and Marvel [21], AP5 [11], and Merlin [19] use conditions as a basis for inferencing. In the absence of explicit consistency management, product consistency is assumed to be achieved implicitly through process correctness. Several process languages, including AP5, APPL/A [39], Marvel, and Merlin, do provide consistency management mechanisms, though all are more limited than the model we have proposed. AP5 handles consistency failures by setting a flag on the inconsistent data; if a particular application cares, it can check the flag, but there is no enforcement mechanism in place to manage inconsistent objects. APPL/A facilitates the definition and enforcement of predicates over relations, but it does not support user-defined actions to be invoked to attempt to repair a violated constraint. Marvel and Merlin use rules to specify consistency constraints, but if the actions fail to repair a consistency violation, the

[4] PL/1 ON conditions do support dynamic enforcement.

violating transaction is aborted. Some process languages also include support for tolerating inconsistency, but these capabilities tend to be somewhat limited (e.g., AP5's consistency flags; APPL/A's mechanisms for suspending the enforcement of a violated constraint, either locally or globally, until it can be repaired).

Active database systems, such as [35, 10, 24, 8], include primitives, typically in the form of event-condition-action (ECA) rules, that can facilitate consistency management. ECA rules are general-purpose mechanisms for detecting the occurrence of some event and responding to it by some action. In fact, the underlying capabilities required to implement consistency management and ECA rules are largely the same, and some researchers have used ECA rules to implement consistency management capabilities. As has been noted by [39, 21, 11, 19], however, the semantics of consistency management and reactive control are fairly different. Consistency management activities are a required part of any computation in which constraints are enforced on objects—they may affect the validity of the computation. The failure to complete the action associated with an ECA rule, however, need not affect the validity of the computation. For example, an ECA rule might be used to send a mail message to a manager when an employee finishes designing a module. Failure to send the mail message is unlikely to have any implications for either the design process or the design artifacts. On the other hand, failure to satisfy well-formedness constraints on the design artifacts impacts the design process and artifacts. For this reason, many process languages with reactive control draw a distinction between consistency management and *automation rules* (e.g., [11, 21, 19, 39]). In these languages, all consistency maintenance activities associated with a process step must be carried out before the step can complete; failure to satisfy constraints typically results in the abort of the associated computation. Automation rules, on the other hand, can be spawned off separately, and their failure does not affect the associated computation.

The difference between ECA rules and consistency management is further evident in the area of inconsistency management. We are unfamiliar with any ECA systems that include mechanisms for specifying and controlling access to inconsistent objects. Depending on the system, it may be possible to implement such semantics manually, but this is not a desirable approach. ECA systems also do not usually include support for handling conditions whose return values are anything other than "true" and "false." Finally, active databases often have the scalability problems that are associated with rule-based systems in general—large collections of rules are difficult to manage and understand. The model of consistency management we proposed helps to address the scalability problem by localizing constraints to the objects to which they pertain.

## 7. Conclusions

Consistency management is an important requirement in many complex applications, but current programming languages and database systems provide very limited support for it. To address this limitation, we defined a consistency management model, based on what we perceive to be the underlying requirements of these applications. These requirements are expressed in terms of both the functional needs as well as cross-cutting concerns that impact how this functionality should be provided. Much of the model has been implemented successfully in the PLEIADES object management system. The focus on our work has been on improved functionality for application programmers, rather than on more quantifiable measures, such as performance. It is very difficult to evaluate functionality, but we attempted to survey users of the system to determine which aspects of the system they used and what they liked and disliked about the system. Based on this evaluation, we have changed some aspects of the model. Other limitations

reported by users can be traced directly to our failure to satisfy some of the functional or cross-cutting requirements in PLEIADES. Overall, clients used most of the capabilities associated with satisfying the functional and cross-cutting requirements and reported that they were quite happy with the support provided. These observations suggest that the requirements we imposed, and the consistency management model we defined, are sound.

Many issues remain to be addressed as future work. First, we would like to make the consistency management model history-sensitive. This is based on the observation that *how* applications reach a particular consistency status matters; for example, knowing that an object is inconsistent is not as useful as also knowing what its status was previously (i.e., whether a particular action actually caused the inconsistency or simply failed to correct it). Second, we are exploring a new approach to addressing the container problem. This approach would incorporate analysis techniques to help guide the selection of appropriate strategies, and specification mechanism that allow developers to state properties of objects that are useful in choosing the best enforcement strategies. Third, we are examining the application of *coupling modes* [29] to consistency management. The presence of coupling modes would permit the decoupling of consistency checking from any associated actions. These modes may be useful, for example, in cases where a reaction to a consistency violation should be deferred to some later time. Third, we hope to explore other implementation strategies and to instantiate the consistency management model for languages other than Ada (in particular, C++ or Java). Our goal in this work would be to evaluate the model in the context of a language that does not include some of the restrictions Ada does (e.g., failure to satisfy the first-class status and identity requirements and very limited dynamic control). Finally, we hope to generalize from the experiences we, and other developers, have had in using PLEIADES and feed those experiences back into the consistency management model.

## Acknowledgments

This work has benefited from the contributions of many people. We are indebted to our Arcadia colleagues and the Avionics Validation and Verification project at TASC for using PLEIADES and for providing us with useful feedback. Lee Osterweil and Stan Sutton have been particularly helpful and have shared their experiences and insights on consistency management in the software process programming domain. We have also benefited from ongoing discussions with Krithi Ramamritham, Jayavel Shanmugasundaram, Arvind Nithrakashyap, and Barbara Lerner.

## References

[1] Kenneth M. Anderson. Personal communication, August 1996.

[2] Timothy Andrews and Craig Harris. Combining Language and Database Advances in an Object-Oriented Development Environment. In Stanley Zdonik and David Maier, editors, Readings in Object-Oriented Database Systems, chapter Object-Oriented Database Systems, pages 186-196. Morgan Kaufmann, 1990.

[3] Malcolm P. Atkinson and O. Peter Buneman. Types and Persistence in Database Programming Languages. ACM Computing Surveys, 19(2):105-190, June 1987.

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

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

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

[7] Naser Barghouti and Gail Kaiser. Concurrency control in advanced database applications. ACM Computing Surveys, pages 269-317, September 1991.

[8] Naser Barghouti and Gail Kaiser. Modeling Concurrency in Rule-Based Development Environments. IEEE Expert, 5(6), December 1990.

[9] Gregory A. Bolcer and Richard N. Taylor. Endeavors: A process system integration infrastructure. In Proc. of the Fourth International Conference on the Software Process, pages 76 - 85, December 1996.

[10] A.P. Buchmann, R.S. Carrera, and M.A. Vazquez-Galindo. A Generalized Constraint and Exception Handler for an Object-Oriented CAD-DBMS. In Proceedings of the International Workshop on Object-Oriented Database Systems, pages 38-49, September 1986.

[11] Don Cohen. AP5 Manual. Technical report, University of Southern California, Information Sciences Institute, March 1988.

[12] R. Conradi, M. Hagaseth, J.-O. Larsen, M. N. Nguyen, B. P. Munch, P. H. Westby, W. Zhu, M. Jaccheri, and C. Liu. EPOS: Object-oriented cooperative process modeling. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, Software Process Modeling and Technology, pages 33 - 70. John Wiley & Sons Inc., 1994.

[13] Reider Conradi, Christer Fernström, and Alfonso Fuggetta. Concepts for evolving software processes. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, Software Process Modeling and Technology, pages 9 - 31. John Wiley & Sons Inc., 1994.

[14] Matthew Dwyer and Lori Clarke. Data Flow Analysis for Verifying Properties of Concurrent Programs. In ACM SIGSOFT'94 Software Engineering Notes, Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, v. 19, n. 5, pages 62-75, December 1994.

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

[16] Volker Gruhn and Rüdiger Jegelka. An evaluation of FUNSOFT nets. In Proc. of the Second European Workshop on Software Process Technology, September 1992. Trondheim, Norway.

[17] Richard L. Hudson. Personal communication, July 1996.

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

[19] G. Junkermann, B. Peuschel, W. Schäfer, and S Wolf. MERLIN: Supporting cooperation in software development through a knowledge-based environment. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, Software Process Modeling and Technology, pages 103 - 129. John Wiley & Sons Inc., 1994.

[20] R. Kadia. Issues Encountered in Building a Flexible Software Development Environment: Lessons from the Arcadia Project. In Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments (SDE5), pages 169-180, Tyson's Corner, VA, December 1992.

[21] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich. Intelligent assistance for software development and maintenance. IEEE Software, 5(3):40-49, May 1988.

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

[23] B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, B. Schiefler, and A. Snyder. Lecture Notes in Computer Science, Vol. 114, chapter CLU Reference Manual. Springer-Verlag, 1981.

[24] Guy M. Lohman, Bruce Lindsay, Hamid Pirahesh, and K. Bernhard Schiefer. Extensions to Starburst: Objects, Types, Functions, and Rules. Communications of the ACM, 34(10):95-109, October 1991.

[25] Joseph P. Loyall, Susan A. Mathisen, Pamela J. Hurley, James S. Williamson, and Lori A. Clarke. An Advanced System for the Verification and Validation of Real-Time Avionics Software. In Proceedings of the Eleventh Digital Avionics Systems Conference, Seattle, WA, October 1992.

[26] D.C. Luckham and F.W. vonHenke. An Overview of Anna, a Specification Language for Ada. IEEE Software, 2(2):9-24, March 1985.

[27] David Maier and Jacob Stein. Development and Implementation of an Object-Oriented DBMS. In Stanley Zdonik and David Maier, editors, Readings in Object-Oriented Database Systems, chapter Object-Oriented Database Systems, pages 167-185. Morgan Kaufmann, 1990.

[28] Eric McCall. Personal communication, July 1996.

[29] D.R. McCarthy and U. Dayal. The architecture of an active data base management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1989, pages 215-224.

[30] T. Owen O'Malley. Personal communication, July 1996.

[31] Leon J. Osterweil. Software Processes are Software Too}. In Proceedings of the Ninth International Conference of Software Engineering, March 1987, pages 2-13.

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

[33] Debra J. Richardson. TAOS: Testing with Analysis and Oracle Support. In Proceedings of the 1994 International Symposium on Software Testing and Analysis, August 1994.

[34] David Rosenblum. Towards a Method of Programming with Assertions. In Proceedings of the Fourteenth International Conference on Software Engineering, May 1992.

[35] David Stemple, Adolpho Socorro, and Tim Sheard. Formalizing Objects for Databases Using ADABTPL. In Proceedings of the Second International Workshop on Object-Oriented Database Systems, pages 110-128, Sept 1988.

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

[37] Stanley M. Sutton, Jr. Preconditions, Postconditions, and Provisional Execution in Software Processes. Technical Report CMPSCI TR 95-77, University of Massachusetts at Amherst, Computer Science Department, Amherst, Massachusetts 01003, August 1995.

[38] Stanley M. Sutton, Jr. Personal communication, May 1996.

[39] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. APPL/A: A Language for Software-Process Programming. ACM Trans. on Software Engineering and Methodology, 4(3):221-286, July 1995.

[40] Stanley M. Sutton, Jr. and Leon J. Osterweil. The design of a next-generation process language. Technical Report CMPSCI Technical Report 96-30, University of Massachusetts at Amherst, Computer Science Department, Amherst, Massachusetts 01003, May 1996.

[41] Peri L. Tarr. Object Management Support for the Construction of Complex Applications. PhD thesis, University of Massachusetts, Amherst, 1996.

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

[43] Peri L. Tarr and Stanley M. Sutton, Jr. Programming Heterogeneous Transactions for Software Development Environments. In Proceedings of the Fifteenth International Conference on Software Engineering, pages 358-369, Baltimore, MD, May 1993.

[44] United States Department of Defense, Washington DC. Reference Manual for the Ada Programming Language, January 1983. Military Standard Ada Programming Language.

[45] Scott L. Vandenberg and David J. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. In Proceedings of the SIGMOD International Conference on Management of Data, pages 158-167. ACM, May 1991.

[46] Alexander E. Wise. IRIS Support Tools User Manual. Arcadia Document 94-03, University of Massachusetts, Amherst, 1994.

[47] Alexander E. Wise. IRIS-Ada Support Tools User Manual. Arcadia Document 95-01, University of Massachusetts, Amherst, 1995.

[48] Alexander E. Wise. Personal communication, July 1996.