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
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 inadequate support for it. To address this limitation, we have defined a consistency management model and incorporated it into the PLEIADES object management system. This paper presents a motivating example that illustrates some typical consistency management requirements and discusses the requirements in terms of both functionality and cross-cutting concerns that affect how this functionality is provided. It then describes the model and some design and implementation issues that arose in instantiating it. Finally, we discuss feedback we have received from users and future research plans.

KEYWORDS
Consistency management, inconsistency management, object management, software engineering environments

1 Introduction

The need to define and maintain *consistency* among objects is a difficult task that arises in many complex applications. One or more objects are 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 environments (SEE). 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. Although many applications and domains can benefit from consistency management, we draw our motivation and examples for this paper from SEEs, a domain with which we are 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 ensuring the meaningful manipulation of objects that are not in consistent states. 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. One reason for this is the degree of diversity of object and consistency semantics—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. Further, some kinds of consistency definitions apply *within* an object, some apply *across* objects, and some apply *globally* [26]. Another factor that makes consistency management difficult is that 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 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 systems support consistency management, we are unfamiliar with any system that supports the wide range of consistency seman-

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.

tics that we have found to be needed in advanced applications, such as SEEs.

This paper presents a model for consistency management and reports on an instantiation of the model and experiences using it. Section 2 provides a small but typical example of a SEE application to illustrate some of the consistency management needs of such applications, and it uses the example to help motivate a set of requirements on consistency management systems. Section 3 describes a model of consistency management that satisfies these requirements. We have implemented much of this model in the PLEIADES object management system [36]. Section 4 discusses various design and implementation tradeoffs involved in instantiating the model for PLEIADES. PLEIADES has been used in the implementation of several applications. Section 5 describes some of these uses and, based on client feedback, evaluates our proposed requirements on, and model of, consistency management. Section 6 examines related research. Finally, Section 7 discusses ongoing and future work.

2 Motivating Example and Requirements

To illustrate some typical consistency management needs in complex applications, we use an example from the Arcadia SEE project [17]. This example includes three types of objects: source code, abstract syntax trees (ASTs), and control flow graphs (CFGs), along with some consistency definitions. Each AST node is associated with the source code from which it was created, and each CFG node is connected to 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-date: To ensure that applications only manipulate ASTs and CFGs corresponding to the current source code version, an up-to-date condition is enforced among these three types of objects. This condition indicates that a set of related ASTs, CFGs, and source are mutually up-to-date either if the source's time stamp is less than the AST's, which is less than the CFG's, or if a manager considers the objects mutually consistent.

No def/use errors: Def/use errors in programs include anomalies like references to undefined variables. 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 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-date condition, on the other hand, can be violated during the normal evolution of code. It will be violated whenever 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 should be permitted. It is expected, however, that once a violation has occurred, consistency will be reestablished by identifying the scope of the change and recomputing the corresponding parts of the source code, AST, or CFG (depending on which object was modified). A failure to reestablish consistency, however, might be considered abnormal and might or might not be permissible, depending on the stage of development. "No def/use errors" is yet another kind of condition. Like the up-to-date condition, "no def/use errors" can be expected to be violated during the normal evolution of a program. Unlike the up-to-date condition, however, it may be possible to leave the "no def/use errors" condition violated, at least for some period of time, during some stages of development. While the CFG is in an inconsistent state, however, a different set of operations might be permitted on the CFG and some of its related objects than are permitted while the CFG is consistent. For example, it should not be possible to release source code whose corresponding CFG fails to satisfy the "no def/use errors" condition, but it may be possible to run a debugger 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 [3, 31, 33] and to ensure the meaningful manipulation of inconsistent objects appropriately.

This simple example illustrates some of the kinds of functionality needed to facilitate the definition and management of object consistency. They also demonstrate some "cross-cutting" requirements, which are constraints on how the functionalities are provided.

2.1 Functional Requirements

After carefully examining the needs of several complex applications, we believe that a consistency management system must provide the ability to do the following:

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

Determine when to detect violations: Different kinds of objects have consistency definitions that require detection of violations at different points. For example, the acyclic invariant on ASTs may not be violated, so

potential violations must be detected before they actually occur; the up-to-date condition can be violated, but violations must be detected and repaired immediately; and violations of the "no def/use errors" condition can be detected as needed to ensure meaningful manipulation while a CFG is inconsistent. These are examples of *operation-driven* detection. Other kinds of detection may also be desirable, including checks initiated at a client's request or at specified process steps. Thus, it must be possible to detect (potential) violations of consistency conditions in different ways, as needed to enforce the required consistency semantics.

Specify enforcement semantics: It must be possible to define appropriate responses to (potential) consistency violations. Responses may range from outright rejection of the violating action, to rolling the affected objects forward into a new, consistent state, to allowing the objects to remain inconsistent and managing them accordingly. In addition, enforcement semantics must consider that an initial repair action may be unsuccessful and that subsequent actions may be required.

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 immediately. Managing inconsistency means being able to detect inconsistencies, ensure the meaningful manipulation of inconsistent objects, and ultimately, reachieve consistency. Tolerating the presence of inconsistency requires support from tools and processes (i.e., they must know how to interact with inconsistent objects) and from the consistency management system (i.e., to provide a means for tools to indicate the level of (in)consistency they can handle and to ensure that tools manipulate objects only in meaningful ways). Ensuring meaningful manipulation may entail, for example, changing the set of operations with which an inconsistent object can be manipulated. In the example, CFGs containing def/use errors can be debugged, but their associated source code cannot be released.

Dynamically change consistency specifications: The consistency conditions that apply to a given object may change over time. For example, "no def/use errors" and "up-to-date" both apply to CFGs, but the former typically is enforced only during release phases, while the latter might be enforced throughout a CFG's lifetime. Similarly, the repair action associated with "up-to-date" may change; during development, recompilation might be invoked automatically, but during releases, a manager might have to approve the change.

2.2 Cross-Cutting Requirements

As discussed in Section 6, the functionality described above can be implemented using capabilities found in many modern programming languages and some databases, but not without extensive programming. One of our goals in this work was to develop a model of consistency management that provides more powerful building blocks than those currently found in programming languages and databases. This model would provide primitive capabilities that are easy to use, but general enough to permit the definition of a wide range of consistency management semantics. Our intention was to provide a consistency management model that subsumes those found in programming languages (e.g., exceptions and assertions) and in databases (e.g., rules and constraints), and that provides useful capabilities not present in existing systems. Towards satisfying this goal, we have further constrained the set of functionalities discussed above with a set of cross-cutting requirements [36] that describe more specifically *how* these capabilities should be provided to produce a flexible, broad-spectrum 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 (e.g., reflection [10]), 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., [7, 31, 17]) demonstrated that different kinds of applications require different programming paradigms and models. *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 modeling of relationships among conditions and/or actions to enable, for example, condition decomposition and the definition of ordering constraints (e.g., "up-to-date" should be checked before

"no def/use errors"). Identity facilitates sharing, which makes it easier, for example, to share enforcement semantics among constraints.

3 Model of Consistency Management

Based on the requirements described earlier, we have defined a model of consistency management. A formal specification of this model is given in [35]. We have incorporated much of the model into the PLEIADES object management system [35, 36] to evaluate the model and to explore implementation considerations. This section describes the consistency management model.

To facilitate the description of this 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 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 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.

3.1 Consistency Conditions

Conditions are used to specify what it means for objects to be consistent. A condition is a function whose return value is a *consistency status* flag that indicates the object's state. 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. **Partial** means that some, but not all, parts of a condition evaluate to consistent. This facilitates the use of decomposition relationships among conditions (e.g., to enable incremental condition satisfaction). **Unknown** means that not enough information is available to evaluate the status of one or more objects [32]. 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 evaluation of a condition. **Consistent** and **inconsistent** are the most commonly used values, though the others 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 object-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. Conditions can be enforced to satisfy any consistency status—e.g., the up-to-date condition on a CFG can be enforced so that it must evaluate to either **consistent** or **partial**. This enables weakened enforcement of consistency definitions, rather than only all-or-nothing. Applications can check, at any time, the consistency status of one or more objects with respect to a given condition. This facilitates types of consistency checking that are not tied to operations on objects (e.g., user-initiated and plan-based).

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 results in the invocation of an operation other than the one specified. **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, if the satisfaction of a condition depends on whether the specified operation actually committed.

The description of when (potential) violations should be

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

identified can be done dynamically or statically, at the per-constraint, per-object, or per-type levels.

3.3 Definition of Enforcement Mechanisms

It is possible to define one or more *actions* to be taken when the consistency status of one or more objects (with respect to a given condition) is found to be unacceptable. By default, violating a condition is assumed to be undesirable, so an exception is raised. This default can be overridden. Actions are essentially procedures and are computationally complete, so any required action may occur in response to a violation. In general, actions are used to prevent or correct a violation, but 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 or could enforce different conditions.

Ideally, once its associated actions have 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 optionally specify *action chains*. These chains may be as long as needed and may be modified dynamically.

3.4 Inconsistency Management

After applying all actions in an action chain, a condition may still be violated. 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 only to be allowed to be **consistent**; thus, if an action chain fails to restore consistency, an 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 specified to cover different consistency status values.

3.5 Using the Model

To demonstrate how this consistency management model could be used, we now revisit the 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 precondition of 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 an exception is raised.

```
condition Is_Acyclic ( The_AST : AST;
                       Edge_Source : AST_Node;
                       Edge_Target : AST_Node ) is
begin
   -- Change is acceptable if target does not
   -- already have a parent.
   if (Get_Parent (Get_Target (Target_Node_For_New_Edge))
      /= Null_AST_Node) then return Inconsistent;
   else return Consistent;
end condition;

action Reject_Update ( The_AST : AST;
                       Edge_Target : AST_Node ) is
begin
   Put_Line  ("Attempted to introduce a cycle");
   raise Attempt_To_Violate_Acyclicity;
end action;
```

By default, **Is_Acyclic** is not enforced on instances of type AST, meaning that this 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 indicating that, at least initially, **Is_Acyclic** should be enforced on AST objects:

```
   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 is very useful for constraints, like "no def/use errors," that apply during more limited periods of time. Note that inter-object constraints are enforced using the same mechanism, but a set of objects is included as the value of the **On_Object** parameter.

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

4 Design and Implementation Concerns

The model of consistency management presented 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 use their favorite development languages and draw on the consistency management extensions. PLEIADES represents one such instantiation of the model, for the Ada programming language.

Instantiating the consistency management model required addressing a number of design and implementation issues; indeed, the model could have been instantiated in any number of ways, depending on the particular decisions. While some issues are specific to an Ada instantiation, many are general issues for any instantiation. This section describes some of these issues and discusses justifications for, and the implications of, some of the decisions we made in implementing PLEIADES. [35] includes a more detailed discussion.

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 semantics can be defined. On the negative side, computationally complete formalisms are difficult to reason about. 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, those formalisms that are more amenable to analysis are not complete, so they restrict the set of possible consistency semantics. In instantiating the consistency model, one must choose an appropriate point on the completeness vs. analyzability spectrum.

As noted earlier, first-class status and identity of objects provides the ability to model relationships among, and constraints on, any kinds of objects. The identity requirement, however, can lead to a fairly serious consistency management system implementation problem. The problem, which we call the *container problem* [35], arises when the consistency status of an object, o, depends on the states of other objects whose states can change independently of o's. A well-known example of the container problem is the dangling reference problem. For example, an application might destroy a node in an AST without realizing that other nodes refer to it. The destroyed node affects the consistency of those that refer to it, according to a referential integrity constraint, but it can be destroyed without the knowledge of the referring nodes. The container problem is pervasive and occurs in many forms, and it is particularly problematic in the context of consistency management. Numerous ad-hoc solutions to this problem have been used, including problem-specific approaches like garbage collection (which addresses only 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 considerable 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 static analysis. This may often be acceptable, but when developers know that they do not require dynamic control (e.g., for enforcing invariants), they should be able to impart this information to the consistency manager. The increased potential for optimization and analysis comes with a loss of flexibility, however; thus, the selection of a point on the optimizability vs. dynamic control spectrum must occur when instantiating the consistency model.

Current status: PLEIADES currently supports much of the model described in Section 3 and addresses many of the functional and cross-cutting requirements.

PLEIADES is implemented as a preprocessor for Ada. Developers describe ADTs using primitives PLEIADES provides, as illustrated in Section 3, and PLEIADES produces an Ada package that provides type and operation definitions for creating, manipulating, and enforcing consistency over instances of those ADTs. Applications 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. Our potential users also felt more comfortable using 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 a restriction on the set of constraint in-

[3] A programming language enhanced with capabilities like consistency management, persistence and concurrency control is typically referred to as a *database programming language* [2].

stantiations that can be described declaratively. Specifically, inputs to PLEIADES describe *types*, not *instances*; thus, type-level constraints can be instantiated statically or dynamically, but instance-level instantiations must occur dynamically. In addition, PLEIADES cannot perform analyses that involve client code, such as identifying unused consistency conditions or statically detecting attempts to enforce constraints on objects to which they do not apply.

PLEIADES directly supports all aspects of the consistency model presented in Section 3, with a few exceptions. First, the set of consistency status values 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 adequate support for inconsistency management. In particular, it does not provide a simple, declarative means of indicating how objects can or cannot be manipulated while they are inconsistent. For most of these limitations, we believe that users can achieve the desired semantics using existing capabilities, and in fact, some users have done so, validating both the need for these semantics and the feasibility of providing them. Achieving these semantics currently requires more programming intervention than we believe is desirable, however, and permits fewer opportunities for analysis and automated support. These restrictions exist because we did not initially recognize the need for these capabilities; user feedback suggested their utility. None are particularly problematic to implement, and we plan to include them in future versions of the system. Finally, the current implementation of PLEIADES only partially satisfies two of the cross-cutting requirements: objects are first-class entities, but conditions and actions are not; and conditions can be enforced only on a subset of types (thus failing to satisfy type completeness fully). 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 for consistency management. We believe that addressing these problems in a more modern language, like Java, is straightforward, both conceptually and from an implementation perspective.

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 obtained directly from several PLEIADES users. The client applications about which we obtained information were a reusable components library, the Arcadia language processing tool set, TAOS (Testing with Analysis and Oracle Support) [29], the Booch Object-Oriented Design process program (BOOD) [34], FLAVERS (Flow Analysis and VERification System) [12], an agenda management system, and an avionics validation and verification system [21]. 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 client had used capabilities resulting from each of the functional and cross-cutting requirements, how closely the provided functionality satisfied the user's needs, and whether current limitations or existing features of PLEIADES caused the user difficulties. We then evaluated 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 complete description of the evaluation appears in [35].

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 the functional requirements, and they made use of all the capabilities associated with the cross-cutting requirements. Typically, problems reported could be traced to a failure to satisfy either a functional or cross-cutting requirement.

As noted earlier, feedback from users led to some changes in the consistency management model. It pointed up other noteworthy items as well, including:

Granularity issues: 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.

Inter-object constraints: 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 static type model, means that only those constraints specified as part of an ADT's definition can apply to its instances. Thus, this user had to group all the ADTs to which inter-object constraints applied in the same specification, rather than separating them appropriately, which reduced modularity.

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

Up-to-date constraints are used to assure percolation of changes among related objects. Applications differ widely in the semantics they attach to failure to repair violations of such constraints, however. Some rely on the success of a repair and cannot continue if repair fails. Others recognize that repair may fail and try alternative repair mechanisms or tolerate the inconsistency.

"Well-formedness" constraints impose type and instance semantics not expressible using standard programming language type models. For example, the AST's acyclic constraint is a well-formedness constraint.

Operation constraints enforce many kinds of full or partial order relationships among invokable entities. For example, **Push** must be invoked on a stack before **Pop**. In most languages, these kinds of ordering constraints must be enforced manually, using checks included in operation implementations. This greatly increases the difficulty of reasoning about, and changing 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 benefits to facilitating their description explicitly.

6 Related Work

Programming languages have useful but limited support for consistency management. 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 restricted to such criteria as bounds checking and erroneous type usage; they do not support complex consistency definitions (e.g., well-formedness and up-to-date). Assertion (e.g., [30, 22, 25]) and exception handling mechanisms (e.g., Ada, CLU [19], and Java) are specialized consistency management mechanisms provided by some languages. Assertions describe invariant conditions of a running program and specify actions to be taken upon detecting violations of invariants. They are embedded within the operations in which they should be checked; thus, they represent static associations of assertions with operations. While enforcement may be turned on and off dynamically, as in Eiffel, the lexical embedding of assertions in operations means that the set of assertions and actions associated with a given operation is fixed statically. Further, instance-level control is very difficult to achieve. Eiffel's support for pre- and postconditions has a similar limitation. In contrast, the model we propose provides both static and dynamic association of conditions with operations, instances and actions, providing full dynamic and instance-level control over consistency management. Exceptions reflect unusual conditions; exception handlers specify actions to be undertaken if one of these conditions arises. Again, lexically embedding code to raise and handle exceptions makes dynamic and instance-level control difficult to achieve. Assertion and exception handling mechanisms usually do not satisfy the cross-cutting requirements for dynamic control over enforcement, or for first-class status or identity of the conditions or actions associated with these mechanisms. They also do not support weakened notions of consistency or accommodate inconsistency—their purpose is generally to preclude violation (particularly in the case of assertions) or to provide a mechanism for repair. If the repair fails, there is no means of describing or enforcing the meaningful manipulation of inconsistent objects.

Many relational database systems support constraints. Their constraint enforcement mechanisms do not, however, satisfy most of the cross-cutting requirements or many of the functional 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., [23]) or programming language kinds of consistency definitions (e.g., [37, 1]), or they support consistency definitions over only a subset of types (typically collection types; e.g., [33, 9]).

Active database systems (e.g., [8, 20, 6]) include reactive control 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 very similar, and some researchers have used ECA rules to imple-

ment consistency management capabilities. As noted by [33, 18, 9, 16], 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—failure to satisfy constraints may invalidate an activity. The failure to complete an action associated with an ECA rule, however, need not invalidate the associated activity. For example, an ECA rule might be used to send a message to a manager when an employee finishes designing a module. Failure to send the message is unlikely to have any implications for either the design process or 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., [9, 18, 16, 33]). 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 aborting the associated step. Automation rules, on the other hand, can be spawned off separately, and their failure does not affect the associated step. In addition, we are not familiar with any ECA systems that include mechanisms for specifying and managing inconsistency. It may be possible to implement such semantics manually in some systems, but this is not a desirable approach. ECA systems also do not usually include support for conditions whose 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 and applications to which they pertain.

Much research has been done in consistency management for software process languages [27]. Some of these languages rely on consistency management support from an associated object management system (e.g., SLANG [4, 5]). Many software process languages incorporate conditions on product state as process control conditions, rather than as product consistency conditions, which assumes that process correctness ensures product consistency. For example, EPOS [11] uses guards and postconditions to affect flow of process control; Grapple [15] and Interact [28] model conditions as goals; Melmac [14] and SLANG use conditional branching; and Marvel [18], AP5 [9], and Merlin [16] use conditions as a basis for inferencing. All of these process languages provide a more restrictive consistency model than the one we have proposed. For example, 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 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 violating transaction is aborted. Some process languages also include support for tolerating inconsistency, but, as noted, these capabilities tend to be somewhat limited.

Consistency management has also been a focus of research in the area of viewpoint management. The ViewPoints system [26], for example, facilitates multi-agent, concurrent requirements engineering by associating a viewpoint with each developer. Each viewpoint represents that developer's own perspective on the requirements under development and the process by which they are being developed. An overriding concern in this work is to support fully distributed development, in which no shared state or central database is required, so each viewpoint contains its own copy of any objects being developed. This work also attempts to address the problem that different viewpoints may contain different (possibly conflicting) views of corresponding objects. Work on consistency of viewpoints in general, and the ViewPoints model in particular, shares several points in common with ours, but it is a fairly different approach to addressing a related problem. It could be argued that the ViewPoints approach is similar to optimistic concurrency control, while our approach is similar to pessimistic. Optimistic concurrency control assumes that conflicts are rare, so it permits them and resolves them by aborting conflicting transactions. ViewPoints permits conflicts between viewpoints and provides mechanisms for their collaborative resolution. Optimistic approaches tacitly assume that loss of work due to conflicts is acceptable. Also, as noted in [13], ViewPoints is limited in its ability to handle "global" consistency conditions, since there is no concept of global or shared state. Pessimistic concurrency control assumes that state may be shared but provides mechanisms that can be used to avoid conflicts. Our model similarly assumes that tools and processes may share state and provides mechanisms for ensuring the meaningful manipulation of objects, including the identification and handling of manipulations that could lead to global or local inconsistencies. It has been our assumption that, particularly where creative effort is involved, the loss of work due to such inconsistency is unacceptable. Both the "optimistic" and "pessimistic" approaches have advantages and disadvantages and are likely to be useful in different situations. Providing support for both would be ideal.

7 Conclusions

Consistency management is an important requirement in many complex applications. Depending on the particular application or domain, the required consistency semantics may vary considerably. Facilitating a wide range of consistency semantics is therefore both desirable and necessary. Current programming languages and databases provide some useful kinds of support, but for a limited range of consistency semantics (and thus, applications). To address their limitations and facilitate the specification and enforcement of a broad spectrum of semantics, we defined a consistency management model, based on what we perceive to be the underlying requirements of complex 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 of 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 surveyed users of the system to determine which aspects of the system they used and what they liked and disliked about it. Based on this evaluation, we have changed some aspects of the model. Many limitations reported by users can be traced directly to PLEIADES' failure to satisfy some of the functional or cross-cutting requirements. 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 may be important—e.g., to determine whether a particular action actually caused an inconsistency or simply failed to correct an existing one. 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 as well as a specification mechanism that allows developers to state properties of objects that are useful in choosing the best enforcement strategies. Third, we are examining the application of *coupling modes* [24] 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. Fourth, we hope to explore other implementation strategies and to instantiate the consistency management model for languages other than Ada (notably, Java). This would help us evaluate the model in the context of a less restrictive language. Fifth, we believe that both "optimistic" and "pessimistic" consistency management mechanisms are important and useful in different circumstances, and while we believe we have included many features that are necessary to support optimistic mechanisms, much of our focus has been on pessimistic mechanisms. We hope to explore the integration of optimistic mechanisms, such as that described in [26, 13], into our model. 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

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 software process programming. We have also benefited from ongoing discussions with Krithi Ramamritham, Jayavel Shanmugasundaram, Arvind Nithrakashyap, and Barbara Lerner.

REFERENCES

[1] T. Andrews and C. Harris. *Combining Language and Database Advances in an Object-Oriented Development Environment*, pages 186–196. Readings in Object-Oriented Database Systems. 1990.

[2] M. P. Atkinson and O. P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, Jun 1987.

[3] R. Balzer. Tolerating inconsistency. In *Proc. 13th Int'l. Conf. on Software Engineering*, pages 158–165, 1991.

[4] S. Bandinelli and A. Fuggetta. Computational reflection in software process modeling: the SLANG approach. In *Proc. 15th Int'l. Conf. on Software Engineering*, 1993.

[5] S. Bandinelli, A. Fuggetta, and S. Grigolli. Process modeling in-the-large with SLANG. In *Proc. 2nd Int'l. Conf. on the Software Process*, pages 75–83, 1993.

[6] N. Barghouti and G. Kaiser. Modeling concurrency in rule-based development. *IEEE Expert*, 5(6), 1990.

[7] N. Barghouti and G. Kaiser. Concurrency control in advance database applications. *ACM Computing Surveys*, pages 269–317, Sep 1991.

[8] A. Buchmann, R. Carrera, and M. Vazquez-Galindo. A generalized constraint and exception handler for an object-oriented CAD-DBMS. In *Proc. Int'l. Workshop on Object-Oriented Database Systems*, pages 38–49, Sep 1986.

[9] D. Cohen. *AP5 Manual*. University of Southern California, Information Sciences Institute, 1988.

[10] R. Conradi, C. Fernstrom, and A. Fuggetta. Concepts for evolving software processes. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modeling and Technology*, pages 9–31. 1994.

[11] R. Conradi, et. al. EPOS: Object-oriented cooperative process modeling. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modeling and Technology*, pages 33–70. 1994.

[12] M. Dwyer and L. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proc. 2nd Symp. on Foundations of Software Engineering*, pages 62–75, Dec 1994.

[13] S. Easterbrook, A. Finkelstein, J. Kramer, and B. Nuseibeh. Coordinating distributed viewpoints: The anatomy of a consistency check. *Int'l. Journal on Concurrent Engineering: Research and Applications*, 2(3):209–222, 1994.

[14] V. Gruhn and R. Jegelka. An evaluation of FUNSOFT nets. In *Proc. 2nd European Workshop on Software Process Technology*, Sep 1992.

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

[16] G. Junkermann, B. Peuschel, W. Schafer, and S. Wolf. MERLIN: Supporting coooperation 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. 1994.

[17] R. Kadia. Issues encountered in building a flexible software development environment. In *Proc. 5th ACM SIGSOFT Symp. on Software Development Environments*, pages 169–180, Dec 1992.

[18] G. E. Kaiser, P. H. Feiler, and S. S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49, May 1988.

[19] B. Liskov, et. al. *Lecture Notes in Computer Science*, volume 114, chapter CLU Reference Manual. Springer-Verlag, 1981.

[20] G. Lohman, B. Lindsay, H. Pirahesh, and K. B. Schiefer. Extensions to starburst: Objects, types, functions, and rules. *Communications of the ACM*, 34(10):95–109, Oct 1991.

[21] J. P. Loyall, S. A. Mathisen, P. J. Hurley, J. S. Williamson, and L. A. Clarke. An advanced system for the verification and validation of real-time avionics software. In *Proc. 11th Digital Avionics Systems Conf.*, Oct 1992.

[22] D. Luckham and F. vonHenke. An overview of ANNA, a specification language for ada. *IEEE Software*, 2(2):9–24, Mar 1985.

[23] D. Maier and J. Stein. Development and implementation of an object-oriented dbms. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 167–185. Morgan Kaufman, 1990.

[24] D. McCarthy and U. Dayal. The architecture of an active database management system. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 214–224, 1989.

[25] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[26] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specifications. *Trans. on Software Engineering*, 20(10):760–773, Oct 1994.

[27] L. J. Osterweil. Software processes are software too. In *Proc. 9th Int'l. Conf. on Software Engineering*, pages 2–13, Mar 1987.

[28] D. E. Perry. Policy-directed coordination and cooperation. In *Proc. 7th Int'l. Software Process Workshop*, 1991.

[29] D. J. Richardson. TAOS: Testing with analysis and oracle support. In *Proc. 1994 Int'l. Symp. on Software Testing and Analysis*, Aug 1994.

[30] D. Rosenblum. Towards a method of programming with assertions. In *Proc. 14th Int'l. Conf. on Software Engineering*, May 1992.

[31] S. M. Sutton, Jr. A flexible consistency model for persistent data in software-process programming languages. In A. Dearle, G. M. Shaw, and S. B. Zdonik, editors, *Implementing Persistent Object Bases—Principles and Practice*, pages 305–318, 1991.

[32] S. M. Sutton, Jr. Preconditions, postconditions, and provisional execution in software processes. Technical Report CMPSCI TR 95–77, Computer Science Department, University of Massachusetts at Amherst, Aug 1995.

[33] S. M. Sutton, Jr., D. Heimbigner, and L. J. Osterweil. APPL/A: A language for software-process programming. *ACM Trans. on Software Engineering and Methodology*, 4(3):221–286, Jul 1995.

[34] S. M. Sutton, Jr. and L. J. Osterweil. The design of a next-generation process language. In *Proc. 5th Symp. on the Foundations of Software Engineering*, Sep 1997. (To appear.).

[35] P. L. Tarr. *Object Management Support for the Construction of Complex Applications*. PhD thesis, University of Massachusetts at Amherst, 1996.

[36] P. L. Tarr and L. A. Clarke. PLEIADES: An object management system for software engineering environments. In *ACM SIGSOFT '93 Symp. on Foundations of Software Engineering*, pages 56–70, Dec 1993.

[37] S. L. Vandenberg and D. J. DeWitt. Algebraic support for complex objects with arrays, identity, and inheritance. In *Proc. SIGMOD Int'l. Conf. on Management of Data*, pages 158–167, May 1991.