OBJECT MANAGEMENT SUPPORT FOR THE CONSTRUCTION OF COMPLEX APPLICATIONS

A Dissertation Presented

by

PERI L. TARR

Submitted to the Graduate School of the University of Massachusetts Amherst in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1996

Department of Computer Science
OBJECT MANAGEMENT SUPPORT FOR THE CONSTRUCTION OF COMPLEX APPLICATIONS

A Dissertation Presented

by

PERI L. TARR

Approved as to style and content by:

Lori A. Clarke, Chair

Leon J. Osterweil, Member

J. Eliot B. Moss, Member

Krithi Ramamritham, Member

Alexander L. Wolf, Member

Don Eric Levine, Member

David W. Stemple, Department Chair
Department of Computer Science
This thesis is dedicated, with love, to my grandmothers, Regina Sage and Jean Tarr, may they rest in peace. They taught me so much about living and making the most of life that it will probably take me a lifetime to understand it all. From them came my belief that I could do anything I wanted to, and it was that belief that led me to enter and complete the Ph.D program. I wish they could have been here to see the result of their legacy to me.
ACKNOWLEDGEMENTS

I couldn’t possibly thank all the people who deserve to be thanked for their roles in helping me to complete this dissertation, but I’ll give it a shot:

- My advisor and mentor, Lori Clarke, who saw potential in me and never gave up in her quest to help me realize it. She has been my role model and, by her example, has shown me the kind of researcher I aspire to become. Every graduate student should be fortunate enough to have an advisor like her.

- Lee Osterweil, whose visions for software engineering have inspired me for years, and whose advice has been invaluable.

- Alex Wolf, who was my first mentor as a new graduate student, and who has always been there to help when I really needed it.

- Eliot Moss, for sharing his ideas about, and experiences with, many different aspects of object management support (and especially those that facilitate and hinder optimization), and for putting so much time and effort into helping me clarify the object management model.

- Krithi Ramamritham and Don Levine, for their very valuable feedback.

- Phil Stocks, who came to my rescue when I had problems specifying the object management framework meta-model in Z. He spent a couple of weeks figuring out what I’d done wrong and how to do it right. The approach that we finally used came from him, and only someone of his knowledge and expertise could have come up with such a clear, clean design. He also taught me an enormous
amount about Z and how to use it the right way in those couple of weeks. I am truly grateful to him for his absolutely indispensable help.

- My mother, father, and brother, for all their support over the years. My mother, in particular, has shown me, during her decade-long battle with cancer, how to face even the worst obstacles in life with courage and dignity, and how important it is to appreciate life to the fullest.

- Sandy Wise, who has been a great friend and collaborator (despite that day in the machine room when he told me, in no uncertain terms, that I was in his way). He has had a lot of impact on much of the work described in this document. He also generously applied his gift for visualizing concepts to produce most of the artwork found throughout this document, and no one does it better than he does. (See for yourself!)

- Owen O’Malley, for being both a terrific friend and a tremendous help technically. He’s done everything from serving as a friendly Pleiades user, to providing tons of extremely valuable feedback and ideas, to producing a storage manager implementation when the one we were using caused performance problems, to providing help with Z. I don’t know what I would have done without him.

- Ken Anderson, whose outstanding feedback and input have significantly helped to guide the development of Pleiades and its predecessors for several years.

- Rick Hudson, for being such an important part of my family (though I still wish he’d remember to pick up seltzer and cat food). He’s always there when you need him. He’s helped me work through many technical problems, and he’s also been nice enough to share with me his outstanding insights into memory management, object viability, and consistency management. (If you’re interested in
memory management and haven’t seen his algorithm [47], you’ve really missed some spectacular work.) And, of course, he’s my very best dessert co-designer and critic.

- Stan Sutton. It would probably take another thesis to thank him for his technical contributions, and a few more beyond that to talk about how much joy and love he’s brought into my life.

- Joyce Vann, who has been my second mother and the best friend anyone could ever hope to have. I would never have come close to making it this far without her.
ABSTRACT

OBJECT MANAGEMENT SUPPORT FOR THE CONSTRUCTION OF COMPLEX APPLICATIONS

SEPTEMBER 1996

PERI L. TARR

B.S., UNIVERSITY OF MASSACHUSETTS AMHERST

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Lori A. Clarke

The definition and long-term management of data in complex systems requires extensive support, including high-level type and behavior modeling, persistence, query-based and navigational access, consistency management, and concurrency control. Traditionally, some of these capabilities have been provided by programming languages (e.g., semantically rich type and behavior models and navigational access), while others have been provided by database management systems (e.g., persistence, queries, and concurrency control). No language or database has provided the full set of required capabilities, however. This has typically required developers to program in multiple paradigms, translating explicitly between “programming language” and “database” models as necessary to use their respective capabilities. The object-oriented database approach has sought to reduce this impedance mismatch in certain areas, but discrepancies still remain.

We have developed a different approach to addressing the object management needs of complex applications. This approach eliminates the dichotomy between
“programming language” and “database” objects, thus allowing the full set of language and database capabilities to be applied equally to all objects. The resulting object management capabilities are provided in a programming language-like manner, sometimes referred to as a “database programming language.” This allows software developers to define objects, their interrelationships, and their behavioral semantics in the same programming language in which they build the systems that manipulate these objects. The object management capabilities can be applied to any kinds of objects these systems may need to define, including non-traditional objects like threads and procedures. The database programming language approach thus reduces the burden on application developers and minimizes application complexity, resulting in more rapid development of more maintainable software.

The database programming language approach raises several challenges, arising in part from the historically different goals of programming languages and databases. Languages are general-purpose and flexible, to support a wide variety of application semantics, while databases impose semantic restrictions to improve performance. Thus, fully integrating programming language and database object management capabilities requires expanding language and database semantics to accommodate capabilities from the other domain. It also requires addressing numerous integration problems that arise when these new semantic models are inconsistent with each other. The result must retain the power of the language and database, and still perform acceptably. In this research, we formally define some of these semantic models, and explore and attempt to address the set of interactions via a prototype implementation and experimental evaluation.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>v</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xiv</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xv</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. OBJECT MANAGEMENT IN COMPLEX APPLICATIONS</td>
<td>8</td>
</tr>
<tr>
<td>2.1 Motivating Example</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Functional Requirements</td>
<td>13</td>
</tr>
<tr>
<td>2.3 Approaches to Object Management</td>
<td>18</td>
</tr>
<tr>
<td>2.4 Cross-Cutting Requirements</td>
<td>21</td>
</tr>
<tr>
<td>3. AN OBJECT MANAGEMENT SYSTEM FRAMEWORK</td>
<td>25</td>
</tr>
<tr>
<td>3.1 High-Level Type and Instance Model</td>
<td>27</td>
</tr>
<tr>
<td>3.1.1 Overview</td>
<td>27</td>
</tr>
<tr>
<td>3.1.2 Formalization</td>
<td>29</td>
</tr>
<tr>
<td>3.1.2.1 Initial Declarations</td>
<td>29</td>
</tr>
<tr>
<td>3.1.2.2 Type Definitions</td>
<td>36</td>
</tr>
<tr>
<td>3.1.3 Satisfying the Cross-Cutting Requirements</td>
<td>53</td>
</tr>
<tr>
<td>3.2 Persistence</td>
<td>54</td>
</tr>
<tr>
<td>3.2.1 Overview</td>
<td>54</td>
</tr>
<tr>
<td>3.2.2 Formalization</td>
<td>56</td>
</tr>
<tr>
<td>3.2.3 Satisfying the Cross-Cutting Requirements</td>
<td>58</td>
</tr>
<tr>
<td>3.3 Navigational and Associative Access</td>
<td>59</td>
</tr>
<tr>
<td>3.3.1 Overview</td>
<td>59</td>
</tr>
<tr>
<td>3.3.2 Formalization</td>
<td>60</td>
</tr>
</tbody>
</table>
3.4 Consistency Management .................................. 60
   3.4.1 Overview .............................................. 60
   3.4.2 Formalization ........................................ 61
   3.4.3 Satisfying the Cross-Cutting Requirements .......... 66
3.5 Concurrency Control ...................................... 67
   3.5.1 Overview .............................................. 67
   3.5.2 Formalization ........................................ 68
3.6 Related Work ............................................. 72
4. PLEIADES: A PROTOTYPE OBJECT MANAGEMENT SYSTEM FOR ADA ........................................ 76
   4.1 Language Features to Support Object Management ......... 76
      4.1.1 Appropriate High-Level Primitive Types .......... 77
         4.1.1.1 Graphs ......................................... 77
         4.1.1.2 Sequences ...................................... 79
         4.1.1.3 Relationships and Relations ...................... 79
      4.1.2 Navigational and Associative Access ............... 85
      4.1.3 Persistence ........................................ 88
      4.1.4 Consistency Management ............................. 92
   4.2 Conclusions and Future Work .............................. 96
5. INSTANTIATING THE PERSISTENCE MODEL ..................... 98
   5.1 Design-Level Issues in Mapping the Framework to Ada ........ 98
      5.1.1 Satisfying the Cross-Cutting Requirements .......... 98
         5.1.1.1 Type-Level Control Over Extent of Persistence .. 101
         5.1.1.2 Instance-Level Control Over Extent of Persistence .. 103
      5.1.2 Naming Issues ...................................... 103
   5.2 Implementation Issues ................................... 105
      5.2.1 Implementation Architecture .......................... 105
      5.2.2 Minimizing Read/Write Costs ......................... 107
      5.2.3 Centralized vs. Decentralized Control over Persistence .... 111
   5.3 Interaction Issues ...................................... 116
      5.3.1 Persistence and TypeExtent .......................... 117

xi
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.6</td>
<td>Agenda Management System</td>
<td>190</td>
</tr>
<tr>
<td>8.7</td>
<td>Chimera</td>
<td>193</td>
</tr>
<tr>
<td>8.8</td>
<td>Chiron</td>
<td>199</td>
</tr>
<tr>
<td>8.9</td>
<td>Summary</td>
<td>203</td>
</tr>
<tr>
<td>8.9.1</td>
<td>Type Model</td>
<td>204</td>
</tr>
<tr>
<td>8.9.2</td>
<td>Navigational and Associative Access</td>
<td>206</td>
</tr>
<tr>
<td>8.9.3</td>
<td>Persistence</td>
<td>208</td>
</tr>
<tr>
<td>8.9.4</td>
<td>Consistency Management</td>
<td>211</td>
</tr>
<tr>
<td>9.</td>
<td>CONCLUSIONS AND FUTURE WORK</td>
<td>217</td>
</tr>
<tr>
<td></td>
<td>BIBLIOGRAPHY</td>
<td>223</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Summary: Cross-Cutting Requirements as Constraints on Functional Requirements.</td>
<td>73</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 A High-Level View of Software Engineering Environments.</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Example of AST, CFG, Def/Ref Annotations, and Dependency Information.</td>
<td>11</td>
</tr>
<tr>
<td>2.3 A Spectrum of Object Management Approaches.</td>
<td>19</td>
</tr>
<tr>
<td>3.1 ADTs in the object management framework. Arrows point from supertypes to their subtypes.</td>
<td>26</td>
</tr>
<tr>
<td>4.1 Partial PLEIADES Type Definitions for Example in Figure 2.2.</td>
<td>82</td>
</tr>
<tr>
<td>4.2 PLEIADES Code for Navigational and Associative Access to Instances of Types Defined in Figure 4.1.</td>
<td>86</td>
</tr>
<tr>
<td>4.3 Using the PLEIADES Persistence Interface.</td>
<td>89</td>
</tr>
<tr>
<td>5.1 Anatomy of a PLEIADES interface package.</td>
<td>106</td>
</tr>
<tr>
<td>5.2 Centralized approach to storage management.</td>
<td>112</td>
</tr>
<tr>
<td>5.3 Decentralized approach to storage management.</td>
<td>113</td>
</tr>
<tr>
<td>5.4 The repository manager abstraction.</td>
<td>115</td>
</tr>
<tr>
<td>6.1 ADT-level concurrency control architecture. (Arrows represent IPC.)</td>
<td>127</td>
</tr>
<tr>
<td>6.2 Storage manager level concurrency control architecture.</td>
<td>131</td>
</tr>
<tr>
<td>6.3 A combined ADT/storage-level concurrency control architecture.</td>
<td>136</td>
</tr>
<tr>
<td>7.1 The container problem.</td>
<td>147</td>
</tr>
<tr>
<td>8.1 IRIS representation for an assignment statement.</td>
<td>163</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

The use of software systems is becoming pervasive throughout society. As software systems are used to support wider ranges of activities, they are rapidly becoming extremely large, highly complex entities that are increasingly difficult to construct, understand, and maintain. Facilitating the construction and long-term maintenance and evolution of complex software systems is therefore crucial for software engineering.

A significant impediment to the construction and evolution of complex applications is inadequate support for the management of the large quantities of long-lived, evolving, highly structured and interrelated objects that often characterize these systems. Broadly speaking, an object is an entity containing state (or value) that an application manipulates to perform its task. Objects may be anything from traditional data items, such as integers, records, and graphs, to non-traditional entities like operations, modules, run-time state, etc. Object management systems facilitate the definition, manipulation, and evolution of objects. Because the management of objects plays such a significant role in the lives of many applications, a suitable object management capability directly supports the construction and evolution of complex systems.

Complex applications and the objects they manipulate have a number of characteristics that affect the definition of any object management systems that support them. These applications create and manipulate large numbers of very complex, semantically diverse objects. The objects may be unstructured, such as files, to highly structured, such as abstract syntax graphs (ASGs), and they may range from coarse-
to fine-grained. Many objects are subject to complex interrelationships and interdependencies. For example, a node in a control flow graph (CFG) is related to, and dependent on, the ASG node from which it was created—when the ASG node changes, its corresponding CFG node has to change as well. Some objects may be transient, such as an intermediate result of a computation, while others, like a system design specification, may continue to exist (and potentially evolve) for long periods of time. Multiple complex applications may have to share objects; they may do so competitively, where applications obtain exclusive access, or cooperatively, where the tools collaborate on the development of objects. Applications may access objects for long or short periods of time. Both the applications and objects may be distributed across a network. Finally, complex applications are characterized by pervasive dynamism—the ways in which they define and manipulate objects are affected by interactions with clients, the results of computations, and the state of the run-time environment.

Thus, the definition and long-term management of objects in complex applications requires extensive support, including high-level type and behavior modeling, persistence, query-based and navigational access, consistency management, and concurrency control. Traditionally, some of these capabilities have been provided by programming languages (e.g., semantically rich type and behavior models and navigational access), while others have been provided by database management systems (e.g., persistence, queries, and concurrency control). No existing language or database has provided the full set of required capabilities, however. Thus, developers have typically had to employ both languages and databases to satisfy their object management requirements. This has typically required developers to program in multiple paradigms (e.g., an object-oriented paradigm in C++ [36] and a relational paradigm in Oracle), translating explicitly between “programming language” and “database” models as necessary to use their respective capabilities. The more recent object-
oriented database approach has sought to reduce this impedance mismatch in certain areas, but discrepancies still remain.

In examining why existing object management approaches fail to satisfy the needs of complex applications, we discovered that many of the limitations occur at the database/programming language boundary. First, some database capabilities cannot be applied uniformly to all programming language objects, and some programming language capabilities cannot be applied uniformly to all database objects. For example, queries can be executed only on persistent objects (i.e., “database” objects) in Matisse [104], and several kinds of programming language objects (e.g., arrays, threads, procedures) cannot be made persistent in [109]. Second, the semantics of references among objects are often different on one side of the boundary versus the other. For example, while references among non-persistent objects are well-defined in ObjectStore [79], the semantics of references between persistent and non-persistent objects are undefined. Third, the use of independent data models and execution engines in databases and programming languages means that applications are developed using side-by-side spaces. This typically results in significant semantic impedance mismatches. For example, different programming languages and databases employ different scoping models (lexical, static, or dynamic), exception models (e.g., resumption vs. termination semantics and propagation semantics), object lifetime semantics (e.g., explicit destruction vs. automatic and initialization/finalization semantics), parameter passing semantics (e.g., available modes may include any of in, out, in-out, and value), thread behaviors, etc. Further, because the programming languages and databases are independent entities, the application developer may have to perform coordination tasks manually (e.g., mark persistent objects as modified in Versant [67] and provide mappings between “database” and “programming language” representations of objects in ONTOS and Matisse [67, 104, 5]). Overcoming these semantic impedance mismatches can be quite costly, in terms of its impact on both development
time and application complexity. Thus, the existence of the database/programming language boundary in any object management approach is a significant limitation in the development of complex applications.

In contrast, we have pursued an approach to object management that facilitates the construction and evolution of complex applications by eliminating the dichotomy between “programming language” and “database” objects, thus allowing the full set of language and database capabilities to be applied equally to all objects. The resulting object management capabilities are provided in a programming language-like manner, sometimes referred to as a “database programming language.” This allows software developers to define objects, their interdependencies and interrelationships, and their behavioral semantics in the same programming language in which they build the applications that manipulate these objects. The object management capabilities then can be applied to any kinds of objects these applications may need to define, including non-traditional objects like threads and procedures. This achieves what we believe must be the goals for object management support for complex applications:

- **Orthogonality**: Any object management functionality can be applied to any kind of object. Orthogonality also means that object management functionalities are not coupled; for example, a set can be queried whether or not it is persistent.

- **Flexibility**: The object management system should be expressive enough to facilitate the definition of any kind of object semantics and behaviors that are required for a particular application.

- **Ease of use**: It should be straightforward for a client to define and use any objects, behaviors, and object management capabilities. For example, while it is possible to model ordered lists using relations, doing so is non-trivial and costly, in terms of comprehensibility, maintainability, and performance.
We believe that in satisfying the orthogonality, flexibility, and ease of use goals, the database programming language approach reduces the burden on application developers and minimizes application complexity, resulting in more rapid development of more understandable, maintainable, evolvable software.

Our goal in this work was to define appropriate object management support for complex applications. We recognize, however, that alternative programming paradigms and implementations might exist that could equally achieve the orthogonality, flexibility, and ease of use requirements, and that requirements specific to a particular domain (e.g., real-time constraints) might dictate the selection of one model or implementation over another. Thus, we specifically did not want to overrestrict the range of permissible semantic models, programming paradigms, or implementation strategies. Instead, we have chosen to produce a framework that describes classes\(^1\) of object management functionality that satisfy the orthogonality, flexibility, and ease-of-use requirements. We have hypothesized the existence of a set of constraints which, when imposed on the set of functional requirements, would result in general models of object management capabilities that achieve these goals. We refer to this set of constraints as cross-cutting requirements, since they represent constraints on all of the functional requirements. Applying the cross-cutting requirements to the functional requirements results in a language-independent object management framework that specifies classes of object management systems that facilitate the construction and evolution of complex applications. Database programming languages may then be constructed by instantiating the framework for a particular language.

The database programming language approach raises several challenging issues, arising, in part, from the historically different goals of programming languages and databases. Languages are general-purpose and tend to favor flexibility, to support a wide variety of application semantics. Databases, on the other hand, favor high

\(^1\)“Classes” is used throughout this document in the general sense, to mean “categories” or “groups,” not in the object-oriented type model sense, unless otherwise noted.
performance to enable the effective manipulation of very large collections of data. To improve performance, database systems typically impose semantic restrictions that facilitate optimization. Thus, fully integrating programming language and database capabilities requires expanding language and database semantics to accommodate capabilities from the other domain. It also requires addressing numerous integration problems that arise when these new semantic models are inconsistent with each other. The result should retain the power of the programming language and database and still perform acceptably.

This thesis provides several contributions. First, it defines what we believe to be an appropriate set of cross-cutting requirements and applies them to a set of required object management functionalities, including high-level types, persistence, concurrency control, consistency management, and navigational and query-based access. It then defines an object management framework describing classes of functional models that we hypothesize to achieve the orthogonality, flexibility, and ease-of-use goals and, thus, to provide appropriate object management support for the construction and evolution of complex systems. Second, it defines a mapping from the framework to a prototype object management system for the Ada programming language, a database programming language called Pleiades. We performed this instantiation with several goals. One was to demonstrate the feasibility of implementing the semantics described by the framework. Another was to provide ourselves with a vehicle for evaluating experimentally the cross-cutting requirements, our hypotheses about the utility of object management models that satisfy those requirements, and the performance of such models. Thus, the final contribution of this research is an informal evaluation of the prototype and of the hypotheses underlying the research. The evaluation includes some results and analyses of the use of Pleiades in a number of complex applications, both academic and industrial.
The rest of this document is organized as follows. Chapter 2 characterizes complex systems, using examples taken from the software engineering environment domain. It also describes and motivates the set of object management functionalities such systems require and a set of cross-cutting requirements. Chapter 3 presents the framework that describes classes of object management functionality that satisfy the cross-cutting requirements. We hypothesize the resulting models to provide appropriate support for constructing and evolving complex applications. Chapter 4 then presents Pleiades, a our prototype instantiation of the framework elaborated in Chapter 3.

Instantiating the framework requires making a number of design and implementation decisions and tradeoffs. Thus, Chapters 5 through 7 talk in more detail about some of the design, implementation, and functionality interaction issues that arose in producing the Pleiades prototype. Chapter 8 describes some academic and industrial use of Pleiades, and it uses those applications to evaluate the prototype, the framework, and our underlying hypotheses about object management support for complex applications. Finally, in Chapter 9, we discuss some conclusions and future work.
CHAPTER 2
OBJECT MANAGEMENT IN COMPLEX APPLICATIONS

In this chapter, we discuss the requirements on object management systems for complex applications. To help motivate these requirements, the chapter begins by describing some tools taken from a fairly representative complex application domain—software engineering environments, and in particular, from the Arcadia environment [117]—and the object management capabilities they require. Section 2.1 presents these tools in a running example used throughout the rest of this document. Using this example, Section 2.2 discusses the specific object management functionalities that facilitate the development and evolution of complex applications. Section 2.3 gives an overview of some approaches that have been taken to providing these kinds of capabilities. Section 2.4 then presents and motivates a set of cross-cutting requirements that we believe constrain the specification of the required object management functionalities in ways that make these capabilities appropriate for supporting complex applications.

2.1 Motivating Example

Software engineering environments, as depicted in Figure 2.1, facilitate the development, maintenance, and long-term evolution of software systems [117]. A software engineering environment helps to ensure the production of high-assurance software systems by providing tools that support various aspects of software development and maintenance (e.g., to produce designs and code, to perform various kinds of analyses on software, and to assist in the testing process), infrastructure components...
Figure 2.1 A High-Level View of Software Engineering Environments.

(e.g., to support aspects of software development such as metrics gathering, communication, data storage, and user interface development), and process programs to support the development and maintenance processes (e.g., by notifying developers of relevant product changes, guiding development activities, and supporting project management). Thus, software engineering environments are populated by collections of complex tools and software processes that aid developers and managers in creating and managing many different kinds of complex objects.

Different kinds of tools, software processes, and people may have fairly diverse notions of what constitutes an “object.” For example, a developer might view “objects” as being artifacts of the software engineering process, which represent pieces of the system under construction. These artifacts might include requirements specifications, designs, code, analysis results, and test suites. To a manager, who defines and controls the engineering process, however, “objects” might be process descriptions and
running instances, tools, resources, schedules, etc. Thus, the kinds of objects that different applications create and manipulate can vary quite widely.

Whatever kinds of objects they manipulate, tools and processes all interact with potentially large numbers of complex objects. The applications, and the objects they manipulate, share some important characteristics. In particular, the objects

- may fall anywhere on a spectrum from unstructured, like files and sets, to highly structured, like requirements specifications and abstract syntax graphs.

- may be subject to access at any level of granularity, from fine-grained to coarse-grained.

- are usually highly interrelated and interdependent. For example, pieces of design modules are related to the requirements they satisfy; pieces of code are related to the parts of designs they implement; and test cases are related to the requirements, designs, and code they evaluate. The interdependencies among objects mean that a change to one object may have far-reaching implications for others, and these changes may cascade.

- have diverse semantics. As noted above, almost any entity may be an object in some application’s view—traditional entities, like arrays and stacks, to non-traditional entities, like tools, processes, operations, rules, run-time state, etc. This means that there may be a very weak distinction between what has traditionally been separated into “data” and “code,” which leads to very diverse object semantics and behaviors.

- may live for long periods of time, such as requirements specifications and code, or may be transient, as are intermediate results of computations.

The applications that use these objects have several characteristics in common as well. The tools and software processes in the software engineering environment
must share the interconnected objects. This sharing may occur *competitively*, where some applications may have to wait for others to finish accessing objects before they can manipulate those objects, or *cooperatively*, where collections of applications work together to develop or evolve objects. Some applications may need to access objects for long periods of time, and they may involve a combination of automated and creative work that cannot be recovered readily in the event of software or hardware failures. The applications and the objects may also be distributed. Finally, the applications are characterized by their dynamic behaviors. Many applications must make decisions about their treatment of objects based on information that may not be available statically—for example, user input or application execution results.

This example presents a high-level picture of a fairly representative complex system domain and provides a sense of both the kinds of objects that might be defined and some of the object management capabilities needed to manipulate them.

To motivate more fully the set of required object management capabilities, we now describe a subset of the capabilities provided by some Arcadia tools [17, 111, 87]. These tools create and maintain four major data structures, depicted in Figure 2.2.
based on source code: abstract syntax trees (ASTs), control flow graphs (CFGs), def-
inition and reference (def/ref) annotations, and dependency information [83]. Each
node in a CFG points to the root of the AST subgraph that elaborates the statement
associated with the CFG node. To facilitate analysis, def/ref information is derived
from an AST and associated with the appropriate node in the corresponding CFG.
Based on the def/ref annotations and the structure of the CFG, dependency informa-
tion is associated with CFG nodes. The kinds of dependency information used here
are data dependence, control dependence, and syntactic dependence. A node \( n \) is data
dependent on a node \( m \) if and only if there is a definition (assignment) of a variable
\( v \) at \( m \) that reaches a reference to \( v \) at node \( n \). A node \( n \) is control dependent on
a node \( m \) if and only if there exists a path from \( m \) to \( n \) that does not include the
immediate forward dominator of \( m \). A node \( n \) is syntactically dependent on a node \( m \)
if and only if it is either control or data dependent on node \( m \).¹ A program fragment
and the resulting AST, CFG, def/ref annotations, and dependency information are
shown in Figure 2.2.

Separate tools build each of these four data structures in turn. A front-end tool
accepts source code and creates an AST. A CFG builder uses the AST to create the
corresponding CFG. The def/ref annotator uses the AST to derive the definition and
reference information that is associated with each node of the CFG. The dependency
builder uses both the CFG and the def/ref annotations to construct dependency
information. A developer might decide to change the source code either by making
a change to the actual source and resubmitting the code for reanalysis or by directly
editing a visual depiction of the AST or CFG. In either case, when such changes occur,
each tool associated with an affected data structure is notified so it can recompute
the appropriate information.

¹Only informal definitions are needed here. The interested reader should refer to [83].
Several tools in the environment allow developers to change the source code. Manipulation of source can occur textually, using an editor like *emacs* or *vi*, or visually, by editing a graphical depiction of the abstract syntax graph or control flow graph. When some representation of the source code is changed, the semantic interconnections among these different representations mean that the scope of the change must be identified and propagated appropriately to the other representations, thus keeping the representations mutually consistent. The changes may also have ramifications downstream for the data definition-reference and data and control flow dependency information, which may have to be recomputed in part or in full.

Other tools in the software engineering environment use these data structures to provide users (developers and/or maintainers) with information about the software system they are developing or maintaining. For example, a data flow analysis tool might be employed to detect anomalous sequences of events [80] using the CFG and def/ref annotations. A cross reference tool might use def/ref annotations to answer users’ questions about a program under development, such as where a variable is referenced or declared. A program maintenance tool could use dependency information to determine which procedures would be affected if a particular statement was modified [64].

### 2.2 Functional Requirements

An analysis of some of the characteristics of complex applications and objects suggests a number of necessary object management capabilities, as described below.

**Appropriate high-level type models:** The presence of complex, structured, fine-grained data, both traditional and non-traditional, indicates a need for a rich, high-level type model to facilitate the definition of any kind of object. Further, the diversity
of object semantics suggests that an appropriate type model must incorporate a highly
descriptive semantic and behavioral model and a rich set of manipulation operations.

Certain classes of abstract data types are ubiquitous in many complex applications. These include graphs [20, 28], varying-length sequences, relations and relationships [109]. Graph objects occur, for example, in the form of abstract syntax graphs, control flow graphs, and dependence information graphs, as seen in Section 2.1. The representation of ordered lists of objects, such as the sections of a document or the list of operands of an AST node, are easily captured through the use of varying-length sequences. Finally, the objects defined within a software engineering environment do not exist in isolation—they are typically connected to a variety of other objects. For example, control flow graphs are connected to the abstract syntax graphs from which they were created, and dependence information is connected to the control flow graph from which it was derived. Relationships are n-ary connections among objects. They are a natural type for representing the interconnections among objects. Relations are collections of relationships. They are useful for gathering together related collections of relationships. In the above example, a relation of all the def/ref annotations can be used to answer cross-reference type queries.

Navigational and associative access: Different kinds of applications need to traverse structured data differently. For example, a data flow analysis tool that determines if any variables are referenced before being defined would traverse a CFG navigationally by following the connections from one node in the graph to another. An analysis tool that reports on all the locations where a selected variable is referenced or defined needs to traverse the annotated CFG associatively by querying the structure to determine all the nodes at which variables were defined or used. Some of the objects created in software engineering environments tend to be traversed only navigationally or only associatively, while others, like a CFG, may have to be traversed both ways.
**Persistence:** Some of the objects created during the execution of an application may have to continue to exist for arbitrarily long periods of time (e.g., requirements specifications, which must persist throughout the lifetime of a software system), while others may be transient (e.g., an intermediate result of a computation). In our example, the AST, CFG, def/ref annotations, and dependence information might all persist as long as the program being developed exists, whereas information about the statements that were exercised on the last execution might be of transient interest.

An object management system should also provide a means of determining when objects are no longer useful or meaningful, so that they may be disposed of appropriately. Again returning to our example, if the abstract syntax graph contained errors and was replaced by a semantically correct graph, the erroneous abstract syntax graph, and all the information derived from it, might become useless and could potentially be archived or destroyed.

**Consistency management:** Defined in its most general sense, consistency management is the process of keeping one or more entities in a state that satisfies some condition. The interrelatedness of, and interdependencies among, objects suggests a need for a consistency management capability, which would be used, for example, to keep a CFG for a module up-to-date with respect to the AST for that module. Consistency management comprises

- the definition of consistency conditions,
- the identification of (potential) consistency violations, and
- the prevention of consistency violations, or the reestablishment of consistency following violations.

To facilitate the construction of complex applications, a consistency management mechanism must support complex consistency definitions and permit a range of ap-
approaches to consistency violation-detection and reestablishment. For example, given the above consistency definition for CFGs, consistency management might be done lazily. In this case, changes to the AST could be permitted to occur as necessary. The consistency management mechanism could then identify the consistency violation after it had already occurred (e.g., the next time the CFG is accessed), and then reestablish consistency by invoking the CFG builder to recreate the CFG as necessary. For other kinds of objects, the violation of a consistency definition may be considered erroneous; these kinds of violations must first be detected and then corrected, either by undoing the changes or performing some other corrective action, before proceeding. Further, different mechanisms for consistency definition, violation identification, and consistency reestablishment may be applicable to a given object during different stages of development and maintenance. For example, it may be acceptable to violate the consistency constraint on the CFG object during software maintenance activities, but it should not be possible to do so while the software is being released. A suitable consistency management mechanism must facilitate the description of a wide variety of consistency definition, violation detection, and reestablishment mechanisms.

We note that the mechanisms that support consistency management are also used to support reactive control, which is the initiation of an action in response to the satisfying of a predicate. The predicate may be satisfied, for example, by the invocation of an operation, when a change in the state of some entity occurs, or when some point in the software life cycle is reached, and the action that may be initiated in response to the satisfying of the predicate may be arbitrarily complex (e.g., send mail to a developer or run an analysis tool). Reactive control is extremely important in many kinds of advanced applications, including software engineering environments, where, for example, it permits the automatic invocation of activities at times when they are known to be necessary, thus relieving developers of these tasks.
Support for advanced transaction management: Transaction management facilities provide support for concurrency control and failure recovery. The need for these capabilities is pervasive. This is illustrated by the software engineering environment example, in which teams of developers work concurrently, for long periods of time, to produce and maintain software systems [13]. A developer’s work must be protected both from conflicting concurrent access by other developers and from hardware and software failures. Thus, object management for complex applications must include support for transaction management.

Some advanced applications place some challenging demands on transaction management capabilities. It must be possible in some cases for developers or tools to work in isolation until they complete certain tasks—in our earlier example, a single tool creates an AST, and no other tool must be allowed to manipulate that AST until its creation is complete. In other cases, multiple developers or tools may have to cooperate to complete a particular task—for example, a number of special-purpose editors, analysis, and testing tools might have to cooperate to help developers to locate and repair errors in modules. A concurrency control mechanism must therefore support both isolated and cooperative development; support for cooperative development should facilitate simultaneous access to shared data when appropriate. Further, since software development is time-consuming, objects may be in use by one or more persons or tools for long periods of time, and the work done is usually costly to reproduce in the event of failure. Both concurrency control and failure recovery support must accommodate these kinds of long-lived activities. It is not acceptable for a developer’s work to be lost, either because a concurrency controller discovers a concurrency control problem and resolves it by aborting the work of one or more developers, or because a system crash occurs and the recovery mechanism discards the developer’s incomplete work.
Other capabilities: Several other kinds of object management functionalities are also important. For example, since objects, and their definitions, may change over time, support for evolution is necessary; version and configuration management are required to track changes to objects and to ensure that tools access mutually consistent collections of objects; and support for distribution is needed to facilitate the development of both distributed applications and distributed objects. Because of the magnitude of the task of supporting the full range of necessary object management capabilities, it is beyond the scope of this thesis. Instead, we have chosen to focus on a core set of required capabilities (i.e., those described earlier in this section) and use those to evaluate our hypotheses about object management support for complex applications. Examination of other required object management capabilities is left for future work.

2.3 Approaches to Object Management

The set of required object management functionalities described above is fairly broad. Subsets of these capabilities are typically found in one of two kinds of systems: programming languages, which usually incorporate rich descriptive mechanisms, navigational access, and primitive concurrency control, and databases, which provide persistence, concurrency control, and resiliency. In general, however, neither programming languages nor database systems have supported the full set of required capabilities.

This suggests that, while the set of requirements could be satisfied in any of several ways, two basic approaches are prevalent: extend databases to include the programming language-like functionalities they lack, or extend programming languages to include missing capabilities from databases. If we take databases and programming languages as starting points for supporting the object management needs of large-scale applications, they actually represent the endpoints of a fairly continuous spectrum.
of approaches to object management, as depicted in Figure 2.3. As we move toward the center of the spectrum from the database side, systems incorporate more and more programming language-like features, while systems closer to the center from the programming language side have more database-like features.

The leftmost point of the spectrum includes relational databases, such as Sybase and Oracle. Extended relational databases, like Starburst [62] and Postgres [102], are relational databases that can incorporate any of a collection of somewhat programming language-like capabilities, including richer set of types, object identifiers for some or all types of objects, and rules. Object-oriented databases, like O2 [33] and ObjectStore [57], replace the relational type model with an object-oriented type model. Object-oriented databases vary widely in their capabilities, but they usually include richer type models, similar to those found in many modern programming languages.

At the rightmost point of the spectrum are “pure” programming languages, such as C++ [103] and Smalltalk [42]. These incorporate rich type and semantic models and may include fairly low-level concurrency control (e.g., semaphores and monitors) and persistence (e.g., files) mechanisms. Persistent programming languages, like Persistent Modula-3 [45], are general-purpose programming languages that provide object persistence. Napier88 [72] is an example of persistent programming language that
also supports concurrency control and resiliency, so it falls to the left of persistent programming languages on the spectrum.

The center point of the spectrum represents what are often referred to as “embedded languages,” which are essentially thin-layer interfaces to database capabilities from programming languages, usually wrapped as C libraries. No actual semantic integration between database and programming language functionality occurs, as with most other approaches represented on the spectrum—this approach simply gives programs access to objects defined in the database. Similarly, some database systems provide the ability for database applications to access programming language objects or capabilities (e.g., via foreign functions). Some examples of this approach are Pascal/R [92] and Postgres [101].

Programming languages and databases have very different, and often competing, goals, however, which tends to cause problems for all approaches that attempt to integrate database and programming language functionality. Programming languages are intended to be flexible, general, semantically rich, and highly descriptive, to support a wide range of application semantics. Databases, on the other hand, are intended to support the manipulation of very large spaces of data. This tends to put efficiency at a premium. To improve performance, database systems often enforce semantic restrictions to facilitate optimization. Thus, different approaches on the spectrum have different cost/benefit tradeoffs in terms of expressiveness vs. efficiency. This means that some points on the spectrum may be more suitable than others for different domains.

This raises the question: what is the best object management approach for facilitating the construction of complex applications? Given the required set of object management capabilities, we believe the best approach is what we will refer to as a database programming language. We use this term somewhat differently from others (e.g., [1]). In particular, a database programming language represents a complete,
seamless integration of “database” and “programming language” capabilities. The approach starts with a general-purpose programming language, extends the language’s functionality so that it can be applied to database objects (e.g., to allow relations and transactions to be used as type constructors in the programming language’s type model), enhances database functionality so that it can be applied to programming language objects (e.g., to allow graphs to become persistent and linked lists to be accessed concurrently), and fully integrates the two sets of expanded capabilities.

2.4 Cross-Cutting Requirements

We have hypothesized that, by adequately constraining the statement of the functional requirements, we could describe classes of object management functionality that achieve this kind of complete, seamless integration, as discussed in Chapter 1. These constraints, to which we refer as cross-cutting requirements, are described and motivated below.

Completeness: Applications can be arbitrarily complex. This complexity can be occur along several dimensions; for example, the types of objects they describe, the kinds of computations they perform, and the collections of resources they access. Because applications can be arbitrarily complex, an object management system that facilitates their construction cannot impose descriptive limitations. A requirement for completeness is, therefore, imposed on all object management functionalities. Computational completeness facilitates the definition of arbitrarily complex computations. Type completeness [71] provides the ability to apply any type constructor to any type, including ones created with other type constructors, which facilitates the definition of complex, structured types. Completeness in general maximizes descriptive ability.

Though there can be restrictions based on limited semantics of a particular type constructor.
**First-class status:**  Given the potential diversity of application semantics, almost any entity may have to be treated as an object by some application. For example, while some applications (e.g., compilers and editors) may manipulate records, graphs, and lists as data, others (e.g., run-time systems and software processes) may manipulate functions, rules, applications, resources, and transactions as data. This means that all entities that may be manipulated as objects must be denotable and manipulable. First-class status provides the ability to directly refer to any object and to treat all objects uniformly. The flexibility to pass a type, procedure, or task as a parameter is an example of this requirement.

**Identity:**  Object identity means that an object has a unique name that is distinct from its state [53]. Identity provides the ability for a given object to be referred to by multiple objects, and for a change to a “shared” object to become immediately visible to any objects that refer to it. Thus, object identity also facilitates (de)composition and reuse.

**Dynamic control:**  Many applications are laden with decision points that occur dynamically during the development or maintenance of an application. The flexibility to exert dynamic control over object management capabilities (e.g., to decide which objects will persist or when consistency definitions should be enforced) is therefore important.

**Meta-data:**  To make decisions dynamically, applications require information about their definitions, run-time state, and environment. This kind of information is referred to as “meta-data.” Information about objects, types, and consistency status are examples of the kinds of meta-data that may be required.
**Generality and heterogeneity:** Previous research has demonstrated that different programming languages, persistence models, transaction models, concurrency control models, etc., may be more appropriate for supporting different kinds of applications and different kinds of applications at different parts of their lifetimes (e.g., [113, 50, 13, 106]). Our experience has led us to conclude that no single, high-level functional model will adequately satisfy the diverse object management needs of all applications. Within each of the functional areas of object management, different models are more appropriate for different projects and at different stages of development. Thus, object management functionality must satisfy both *generality* and *heterogeneity* requirements. Generality means that an object management system provides the right primitives to facilitate the definition of useful semantic models or implementation strategies. Heterogeneity means that object management systems must, at minimum, allow alternative models and implementations to coexist peacefully. Taken together, support for generality and heterogeneity in an object management system will allow developers to choose or readily develop the models and implementations that best satisfy their needs, and to vary these according to the demands of their projects.

We note here that one critical aspect of satisfying the requirements for generality and heterogeneity is to achieve a clean *separation of issues*—i.e., to achieve orthogonality. It has often been the case that object management support has coupled some functionalities with others—for example, traditional transaction models have coupled persistence, concurrency control, consistency control, and atomicity. Complex applications in particular require the ability to choose the models of functionality that best support their intended semantics. When coupling of functionality occurs, it is very difficult or impossible for applications to “mix and match” different models of functional support, or even to use only a subset of the available object management capabilities (e.g., the fact that an object may be transient does not imply that an
application that uses this object does not need to enforce consistency constraints on it). Understanding the ways in which different functionalities interact is crucial in achieving a complete separation of issues, and thus, in satisfying the requirements for generality and heterogeneity and ultimately, orthogonality.
CHAPTER 3
AN OBJECT MANAGEMENT SYSTEM FRAMEWORK

In this section, we examine the implications of the cross-cutting requirements for five of the necessary areas of functional support: high-level type models, persistence models, navigational and associative access, consistency management, and concurrency control.

The descriptions of how each cross-cutting requirement constrains each functional requirement are given using a combination of informal, textual descriptions and, in some cases, more formal descriptions. The goal of these formal descriptions is to facilitate the mapping of the described functionality to a programming language and to ensure that the prescribed semantics are provided. We have not presented formalizations in cases where bodies of theoretical research already exist that have formalized the concepts.

As described in Section 1, we have defined an object management system framework that describes classes of object management functionality satisfying the cross-cutting requirements. This framework is specified using a simple, descriptive type model, augmented with Z [97] specifications when formal specifications are helpful in clarifying the intended semantics. The type model is based on those in common object-oriented programming languages, such as C++ [103], Ada 95 [49], and Smalltalk [42], and it allows us to talk about the effect of the cross-cutting requirements in terms of types, attributes, and operations on abstract data types (ADTs) and objects. A type is simply a named set of attributes and operations. An attribute
Figure 3.1 ADTs in the object management framework. Arrows point from super-types to their subtypes.

is a named, typed value. An operation is a named, invokable entity. All objects have object identifiers (OIDs) that uniquely identify them.

The type model we employ is similar to the one used in [14] and includes the following characteristics:

- Support for multiple inheritance. A type inherits all attributes and operations from all of its supertypes and is behaviorally their subtype.

- Support for types as first-class entities.

- Support for identity of types and attributes (since types and attributes are themselves objects). This feature has some very beneficial effects on the descriptive power of the type model, as will be discussed subsequently.

- A self-descriptive meta-model, very similar to that found in Smalltalk, in which each type is an instance of a metatype that describes the type’s properties.

- Full extensibility.
We note that the type model is primarily a notation for discussing the impact of the cross-cutting requirements on the functional requirements. It is not intended either to impose a requirement for an object-oriented type model in object management systems for complex applications, or to define an implementation guide. We selected the object-oriented type model because it included many useful features and semantics, which facilitated the description of how the cross-cutting requirements constrain the functional requirements. It would, however, be possible, and potentially desirable, to produce mappings from this type model to programming languages that are not object-oriented. Further, we recognize that the use of this type model as the basis for an implementation of an object management system would require the specification of additional properties, such as an inheritance conflict resolution policy, constraints on subtype/supertype relationships, the definition of name space semantics, and a query language. In keeping with our goal not to overrestrict the class of object management systems that facilitate the construction of complex applications, we have imposed only those semantics necessary for describing the impact of the cross-cutting requirements on the functional requirements.

Figure 3.1 depicts the type hierarchy. The types shown in the figure are described more fully below. We use an Ada-like syntax to elaborate these definitions, and Z specifications to describe the intended semantics of some of the types and operations described in this section.

3.1 High-Level Type and Instance Model

3.1.1 Overview

To satisfy the cross-cutting requirements, a type model should have the following properties:
Completeness: Any type constructor should be applicable to any type, including one created with any type constructor.\(^1\) It should be possible, for example, to define graphs of relations and relationships between graphs or relations. Type completeness has been defined formally in the literature (e.g., [71]); we employ the same definition.

First-class status and identity: All types and instances of types (where such entities as procedures, functions, and program units are considered to be typed objects) should be first-class entities with identity. This allows all entities to be treated uniformly, and it provides, for example, the ability for any object to be referred to by multiple objects.

Meta-data: Type- and instance-specific meta-data should be provided on any object; this meta-information should permit an application to determine type information about any given entity at any time during its execution.

Dynamic control: It should be possible to create, destroy, and modify instances of any type dynamically. The uniform treatment of types and instances means that it should also be possible to create, destroy, and modify types dynamically. Clearly, certain constraints may have to be imposed to prevent dangerous or undesirable creation, destruction, or modification activities.

Generality and heterogeneity: The type and instance models should readily support the definition of alternative semantics. For example, relations may have set or multiset semantics, and graphs may be defined as connected structures or as collections of nodes and edges. A type model that requires significant amounts of application code to produce commonly used type semantics is inappropriate. Thus, the set of primitive operations defined on all types should be reasonably general and

\(^1\)Note that it should be legal for developers to restrict explicitly the range of types to which a constructor applies.
should be application-extensible. This requirement suggests that it is important to provide a reasonable set of basic type constructors that support the definition of many different semantics, but it is also important to avoid a proliferation of type constructors with only minor semantic differences.

### 3.1.2 Formalization

Satisfying the cross-cutting requirements implies a set of types and operations in the framework. These entities are described below. We also discuss how these definitions satisfy the cross-cutting requirements.

#### 3.1.2.1 Initial Declarations

To facilitate the specification in Z of some of the types and operations in the object management framework, we introduce several core definitions in Z. These definitions are used to model the type hierarchy depicted in Figure 3.1, and they are based on specifications proposed by Stocks [100].

We employ the following Z stylistic conventions throughout this section:

- Given sets are declared using all upper-case letters (e.g., \texttt{ENTITIES}).

- Input parameters in operation schemas are suffixed with a question mark.

- Output parameters in operation schemas are suffixed with an exclamation point.

\[
\text{ENTITY}\]

\text{ENTITY} is used to represent instances of all types. This set is partitioned appropriately to impose type compatibility semantics—e.g., values and objects are distinct subsets of \text{ENTITY}, while operations are subsets of objects. To allow for the specification of behavioral polymorphism semantics in Z, however, all entities must be taken from the same set; thus, \text{ENTITY} encompasses all types of objects.

\[
\text{[OIDs}, \text{STRINGS}]\]
ATTRIBUTE_NAMES : Π STRINGS
TYPE_NAMES : Π STRINGS
OPERATION_NAMES : Π STRINGS
PARAMETER_NAMES : Π STRINGS

These sets are used to represent various kinds of designators. OIDs is the set of object identifiers that are associated with each instance of type Object. ATTRIBUTE_NAMES, TYPE_NAMES, OPERATION_NAMES, and PARAMETER_NAMES represent the set of names given to attributes, types, operations, and operation parameters, respectively. STRINGS is used to represent string values.

[PARAMETER_MODES]

The set PARAMETER_MODES is used to specify the legal set of modes on operation parameters. For example, the definition for PARAMETER_MODES in Ada would include in, out, and in out. The particular set of parameter modes varies among programming languages, and, as the specific modes themselves are not important for purposes of formalizing the framework, we define PARAMETER_MODES as a given set.

<table>
<thead>
<tr>
<th>Types : Π ENTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects : Π ENTITY</td>
</tr>
<tr>
<td>Values : Π ENTITY</td>
</tr>
<tr>
<td>Attributes : Π ENTITY</td>
</tr>
<tr>
<td>Operations : Π ENTITY</td>
</tr>
<tr>
<td>Constraints : Π ENTITY</td>
</tr>
<tr>
<td>Actions : Π ENTITY</td>
</tr>
<tr>
<td>Transactions : Π ENTITY</td>
</tr>
</tbody>
</table>

\langle Objects, Values \rangle partitions ENTITY
Types \cup Operations \cup Attributes \subseteq Objects
Types \cap Operations = \{}
Types \cap Attributes = \{}
Operations \cap Attributes = \{}
Constraints \subseteq Operations
Actions \subseteq Operations
Constraints \cap Actions = \{}
Transactions \subseteq Operations
The axiomatic definitions above define the set of values associated with types Type, Object, Value, Attributes, Operations, Constraints, Actions, and Transaction, which are the types provided in the object management framework (see Figure 3.1). The relationships among these types are also defined appropriately.

\[ Value_{Image} : Values \rightarrow STRINGS \]

Value_{Image} is a function that maps each instance of type Value to a string representations for the value. As will be described in Section 3.1.2.2, images are intrinsic properties of all Values.

<table>
<thead>
<tr>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter_Name : PARAMETER_NAMES</td>
</tr>
<tr>
<td>Parameter_Type : Types</td>
</tr>
<tr>
<td>Mode : PARAMETER_MODES</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter_List : seq Parameter</td>
</tr>
<tr>
<td>Return_Type : Types</td>
</tr>
</tbody>
</table>

\[ \forall P_1, P_2 : Parameter \mid P_1 \in \text{ran Parameter}_{List} \land \]
\[ P_2 \in \text{ran Parameter}_{List} \land P_1 \neq P_2 \Rightarrow P_1.Parameter_{Name} = P_2.Parameter_{Name} \Rightarrow P_1.Parameter_{Type} \neq P_2.Parameter_{Type} \]

Types Parameter and Signature are used to represent operation signatures. The predicate on type Signature states that no operation signature may have two parameters with the same name and type.

\[ Enforce\_Time ::= PREINVOKE \]
\[ \mid PRECONDITION \]
\[ \mid POSTCONDITION \]
\[ \mid POSTEXECUTE \]

The values of type Enforce_Type indicate the times at which an enforced constraint could be checked. These alternatives will be described in Section 3.4.
<table>
<thead>
<tr>
<th>Enforcement_Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check_In_Operation : Operations</td>
</tr>
<tr>
<td>When_To_Check : Enforce_Time</td>
</tr>
<tr>
<td>Action_To_Invoke : Actions</td>
</tr>
</tbody>
</table>

Type **Enforcement_Info** describes information about when and how to enforce a particular constraint. **Check_In_Operation** indicates an operation in which a constraint could be violated (and thus, should be checked). **When_To_Check** indicates when, with respect to the execution of **Check_In_Operation**, the constraint should be checked. **Action_To_Invoke** denotes an operation to be invoked upon (attempted) violation of a constraint. This information is described in detail later in this chapter (Section 3.4).
Object Hierarchy

Entities \( \not\supset \) ENTITY; OID_Of : Objects \( \leftrightarrow \) OIDs
Entity_Type : ENTITY \( \rightarrow \) Types; Type_Name : Types \( \rightarrow \) TYPE_NAMES
Subtype : Types \( \leftrightarrow \) Types
Object_Attributes : Objects \( \leftrightarrow \) Attributes
Entity_Operations : ENTITY \( \leftrightarrow \) Operations
Operation_Name : Operations \( \rightarrow \) OPERATION_NAMES
Operation_Signature : Operations \( \rightarrow \) Signature
Enforced_Constraints, Enforced_Constraints : ENTITY \( \leftrightarrow \) Constraints
Enforced_Constraint_Info : (ENTITY \( \times \) Constraints) \( \leftrightarrow \) Enforcement_Info
Attribute_Name : Attributes \( \rightarrow \) ATTRIBUTE_NAMES
Attribute_Value : Attributes \( \rightarrow \) ENTITY
Attribute_Type : Attributes \( \rightarrow \) Types

(dom OID_Of \( \subseteq \) Entities) \( \land \) (dom Type_Name \( \subseteq \) Entities)
(dom Entity_Type = Entities) \( \land \) (ran Entity_Type \( \subseteq \) Entities)
dom Subtype \( \cup \) ran Subtype \( \subseteq \) ran Entity_Type
dom Entity_Operations \( \cup \) ran Entity_Operations \( \subseteq \) ran Entity_Type
dom Entity_Constraints \( \cup \) ran Entity_Constraints \( \subseteq \) ran Entity_Type
Enforced_Constraints \( \subseteq \) Entity_Constraints
dom Operation_Signature \( \subseteq \) Entities
(dom Operation_Name \( \subseteq \) Entities) \( \land \) (dom Object_Attributes \( \subseteq \) Entities)
dom Operation_Name = dom Operation_Signature
ran Object_Attributes \( \subseteq \) dom Attribute_Name
dom Attribute_Name = dom Attribute_Value
\( \forall O : \) dom Object_Attributes;
\( A_1, A_2 : \) Attributes \( \mid \) \{ \( A_1, A_2 \) \} \( \subseteq \) Object_Attributes\( \mid \) \{ O \} \( \mid \) \( A_1 \neq A_2 \) \( \bullet \)
\( \text{Attribute_Name}(A_1) = \text{Attribute_Name}(A_2) \Rightarrow \)
\( \text{Attribute_Type}(A_1) \neq \text{Attribute_Type}(A_2) \)
\( \forall A : \) dom Attribute_Value \( \bullet \)
\( \text{Entity_Type} (\text{Attribute_Value}(A)) = \text{Attribute_Type}(A) \)
\( \forall T : \) Types \( \mid \) T \( \in \) Entities \( \bullet \) (T, T) \( \not\supset \) Subtype*
\( \forall O : \) Objects \( \mid \) O \( \in \) Entities \( \bullet \) Object_Attributes\( \mid \) \{ O \} \( \mid \) =
\( \text{Object_Attributes} \mid \text{Subtype}^* \mid \{ \text{Entity_Type}(O) \} \mid \)
\( \cup \text{Object_Attributes} \mid \{ \text{Entity_Type}(O) \} \mid \)
\( \forall E : \) Entities \( \bullet \) Entity_Operations\( \mid \) \{ E \} =
\( \text{Entity_Operations} \mid \text{Subtype}^* \mid \{ \text{Entity_Type}(E) \} \mid \)
\( \cup \text{Entity_Operations} \mid \{ \text{Entity_Type}(E) \} \mid \)
\( \forall E : \) Entities \( \bullet \) Entity_Constraints\( \mid \) \{ E \} =
\( \text{Entity_Constraints} \mid \text{Subtype}^* \mid \{ \text{Entity_Type}(E) \} \mid \)
\( \cup \text{Entity_Constraints} \mid \{ \text{Entity_Type}(E) \} \mid \)
\( \forall E : \) ENTITY; C : Constraints \( \bullet \) (E, C) \( \in \) dom Enforced_Constraint_Info
\( \Rightarrow E \in \) Entities \( \land \) C \( \in \) Entities
\( \forall EI : \) ran Enforced_Constraint_Info \( \bullet \)
\( EI.\text{Check_In_Operation} \in \) Entities \( \land \) EI.\text{Action_To_Invoke} \in \) Entities

33
Object_Hierarchy describes the type model that underlies the object management framework. Object_Hierarchy includes the following declarations:

- **Entities**: the set of all existing instances of all types. This set is a subset of the given set ENTITIES, which represents all possible instances of all types.
- **OID_Of**: an injection mapping objects to their OIDs.
- **Entity_Type**: a function mapping entities to their types.
- **Subtype**: represents the “is subtype of” relation, which maps types to their supertypes.
- **Object_Attributes**: a relation mapping entities to the operations defined on them.
- **Operation_Name**: a function mapping operations to their names.
- **Entity_Constraints**: a relation that maps entities to the constraints defined on them.
- **Enforced_Constraints**: a relation mapping entities to the constraints that are currently enforced on those entities. Note that this relation must be a subset of Entity_Constraints.
- **Enforced_Constraint_Info**: maintains information about how each enforced constraint on each entity should be enforced.
- **Attribute_Name, Attribute_Type, Attribute_Value**: functions that map attributes to their names, types, and values, respectively.

We assume standard subtyping semantics in the object hierarchy—i.e., a type has all of the attributes (including operations and constraints) of its parents, and overriding is permissible as long as the new attribute is a subtype of the corresponding
attribute in the parent. Smalltalk [42] defines these semantics precisely, so we do not specify them explicitly in the object hierarchy.

```
Object_Management_Framework

Object_Hierarchy
Meta_Type : Types
Object : Types
Value : Types
Type : Types
Attribute : Types
Operation : Types
Constraint : Types
Action : Types
Transaction : Types
Persistent : ATTRIBUTE_NAMES
Boolean : Types
True : Objects
False : Objects
```

\[\text{Meta_Type} \notin \text{dom Subtype} \]
\[\{(\text{Object, Meta_Type}), (\text{Value, Meta_Type}),
(\text{Type, Object}), (\text{Attribute, Object}),
(\text{Operation, Object}), (\text{Transaction, Operation}),
(\text{Constraint, Operation}), (\text{Action, Operation})\} \subseteq \text{Subtype}\]
\[\forall T : \text{Types} \mid T \in \text{Entities} \land T \neq \text{Meta_Type} \bullet
(\text{T, Meta_Type}) \in \text{Subtype}^*\]
\[\forall V : \text{Values} \mid V \in \text{Entities} \bullet (\text{Entity_Type}(V), \text{Value}) \in \text{Subtype}^*\]
\[\forall O : \text{Objects} \mid O \in \text{Entities} \bullet (\text{Entity_Type}(O), \text{Object}) \in \text{Subtype}^*\]
\[\forall T : \text{Types} \mid T \in \text{Entities} \bullet (\text{Entity_Type}(T), \text{Type}) \in \text{Subtype}^*\]
\[\forall A : \text{Attributes} \mid A \in \text{Entities} \bullet
(\text{Entity_Type}(A), \text{Attribute}) \in \text{Subtype}^*\]
\[\forall O : \text{Operations} \mid O \in \text{Entities} \bullet
(\text{Entity_Type}(O), \text{Operation}) \in \text{Subtype}^*\]
\[\forall O : \text{Objects} \mid O \in \text{Entities} \bullet
\exists A : \text{Attributes} \mid (\text{Attribute_Type}(A) = \text{Boolean}) \land
(\text{Attribute_Name}(A) = \text{Persistent}) \bullet (O, A) \in \text{Object_Attributes}\]
\[
\{\text{Boolean, True, False} \} \subseteq \text{Entities}\]
\[\text{Entity_Type}(\text{True}) = \text{Boolean}\]
\[\text{Entity_Type}(\text{False}) = \text{Boolean}\]

Type \text{Object_Management_Framework} represents a specialization of type \text{Object_Hierarchy} in which all of the specific types defined in the object management framework are declared. Note that the difference between types \text{Object_Hierarchy} and
Object_Management_Framework is that Object_Hierarchy describes the semantics of the type model we employ in defining the object management framework, while Object_Management_Framework adds the declarations of the types shown in Figure 3.1 to Object_Hierarchy. To map the object management framework to a programming language, a developer would further specialize Object_Management_Framework with the semantics of the that language.

3.1.2.2 Type Definitions

**MetaType:** Type MetaType is introduced as the root of the type hierarchy. Type MetaType always exists—it cannot be removed from the type hierarchy. It has no supertypes, and has the following operation:

- function Do_Query ( T : MetaType; Q : Query ) return MetaType;

This operation retrieves information about a type or instance via associative access. Note that, because queries are dependent on properties of the particular types of objects being queried, the signature for a query operation cannot be more specific than this. In addition, since the form of the query is unimportant (and different query formalisms may provide different benefits and disadvantages), the type model does not further specify a query model—one must be provided as part of a mapping of the framework into a programming language.

- function Type_Of ( M : in MetaType ) return Type;

```
Type_Of
∃Object_Management_Framework
E? : ENTITY
T! : Types

E? ∈ Entities
T! = Entity_Type(E?)
```

Returns the type of a given entity.
**Value:** Type `Value` is used to represent entities whose identifiers are the same as their values. Typically, only numeric types (e.g., enumeration types, integers, and floats) are represented as values, but the set of types represented as `Values` is language-specific. Thus, the set of operations defined on type `Value` is limited to the following:

- function `Image_Of (V : in Value) return String;`

\[
\begin{array}{l}
\text{Image Of} \\
V? : Values \\
I! : STRINGS \\
I! = Value_image(V?)
\end{array}
\]

Returns a string representation for a value.

- function `"=" (Left, Right : in Value) return Boolean;`

Determines whether or not two values are identical.

**Object:** Type `Object` is the type of all object types. All objects have a unique object identifier (OID) that uniquely identify them. Type `Object` has one supertype: `MetaType`. We note that we use the Smalltalk metamodel, in which each type is an instance of its own metatype, and the metatype is described as a subtype of type `Type` (which, in Smalltalk, is called `Class`). The name of the metatype of a given type is the name of the type, suffixed with `type`. Thus, type `Object` is an instance of type `Object type`.

To facilitate the specification of type `Object` and its subtypes, many of which define their own `Create` and `Destroy` operations, we include Z schema specifications for two operations that are not actually defined on any type in the type hierarchy: `Create_Object_Types_Shared` and `Destroy_Object_Types_Shared`. These schemas encapsulate those semantics of instance creation and destruction that are common to type `Object` and all its subtypes.
Create\_Object\_Types\_Shared returns a newly created object. The predicate indicates that this new object is given a unique OID (i.e., one that was not already assigned in the function OID\_Of).

Destroy\_Object\_Types\_Shared destroys an existing object. The predicate part indicates that all of the connections between the destroyed object and its associated information (i.e., its attributes, operations, constraints, and OID) are was not already assigned in the function OID\_Of) are eliminated.

The following operations are defined on type Object:

- **function Create** return Object;  

  \[
  \text{Create\_Object} \quad \text{Create\_Object\_Types\_Shared} \quad \text{Entity\_Type}' = \text{Entity\_Type} \cup \{ \text{New\_Object!} \mapsto \text{Object} \}\]

  Creates a new instance of type Object.

- **procedure Destroy** ( Obj : in out Object );
Destroys an object.

Exceptions: Obj is an invalid object (e.g., because it has been destroyed previously).

- procedure Set_Type ( Obj : in out Object; 
  T : in Type );

\[
\text{Set_Type}
\]
\[
\text{\Delta Object\_Management\_Framework}
\]
\[
O? : \text{Objects}
\]
\[
T? : \text{Types}
\]
\[
O? \in \text{Entities}
\]
\[
T? \in \text{Entities}
\]
\[
\text{Entity\_Type}' = \text{Entity\_Type} \oplus \{ O? \mapsto T? \}
\]

Sets the type of object Obj to be T. The semantics of what happens to the values of any preexisting attributes are implementation-specific, as are any restrictions that may be made to the ways in which an object’s type can be changed (e.g., restricting T to be a subtype of Type_Of ( Obj )).

**Type:** Type Type is the supertype of all type definitions. As noted earlier, we employ the Smalltalk metamodel, in which all types are instances of their own metatypes. Type Type is the root of the metatype hierarchy. Its subtypes form a parallel hierarchy to that described under type Metatype. For example, type Metatype has a metatype description, Metatype type, which is a subtype of Type. Since type Metatype is the supertype of type Object, Object’s metatype description, type Object type, is a subtype of Metatype type.

Type Type is defined as an entity that may have attributes (essentially \(<\text{name, type, value}>\) tuples), operations, and constraints (defined later in this section). We
note that, although operations and constraints are specified as separate declarations in type Object_Hierarchy and are added to types separately from attributes, both kinds of entities are simply attributes of Types and their instances. That is, specifying an operation or constraint on a type is equivalent to declaring an attribute on the type whose value is the operation or constraint. We distinguish operations and constraints from attributes in the definition of type Type for convenience, as developers typically manipulate them differently, but the intent throughout is for the operations and constraints that are associated with types to be modeled as attributes.

All type constructors return instances of Type. Type Type has one supertype, type Object, which has the effect of making all types first-class entities with identity. Type Type provides the following operations:

- function Create ( TypeName : String;
                     SuperTypes : set of Type ) return Type;

```plaintext
Create_Type
Create_Object_Types_Shared
Supertypes? : \mathbb{P} \text{Types}
TypeName? : TYPE_NAMES
New_Object! \in \text{Types}
\forall S : \text{Supertypes}? \bullet S \in \text{Entities}
    \begin{align*}
    \text{TypeName}' = \text{TypeName}\cup
    \{ \text{New_Object!} \mapsto \text{TypeName}? \}\n    \text{Subtype}' = \text{Subtype} \cup \{ S : \text{Supertypes}? \bullet (\text{New_Object!}, S) \}\n    \text{Entity_Type}' = \text{Entity_Type} \cup \{ \text{New_Object!} \mapsto \text{Type} \}
\end{align*}
```

Create a type that is a subtype of the given SuperTypes.

- procedure Destroy ( T : in out Type );

```plaintext
Destroy_Type
Destroy_Object_Types_Shared
O? \in \text{Types}
\neg (\exists T : \text{Types} \bullet \text{Subtype}(\{ T \}) = \{ O? \})
\text{Entity_Type} \triangleright \{ O? \} = \{ \}
\text{Subtype}' = \{ O? \} \triangleleft \text{Subtype} \cup \text{Subtype} \triangleright \{ O? \}
\text{TypeName}' = \{ O? \} \triangleleft \text{TypeName}
```

40
Destroy the given type and all its instances, automatically unlinking \( T \) from all its supertypes and subtypes. Note that a type cannot be destroyed if destroying it would leave any of its subtypes parentless. Further, it is not permissible to destroy a type with existing instances.

Exceptions: \( T \) is the last parent of one of its subtypes; at least one instance of \( T \) exists.

- procedure \texttt{Link\_To\_Supertype} ( \( T \) : in out Type; \( \text{SuperType} \) : in out Type );

\[
\begin{array}{l}
\text{Link\_To\_Supertype} \\
\text{Object\_Management\_Framework} \\
T? : \text{Types} \\
\text{SuperType}? : \text{Types} \\
(T?, \text{SuperType}?) \not\in \text{Subtype} \\
(\text{SuperType}?, T?) \not\in \text{Subtype} \\
\text{Subtype}' = \text{Subtype} \cup \{(T?, \text{SuperType}?)\}
\end{array}
\]

Add \( \text{SuperType} \) to \( \text{Supertypes\_Of} ( T ) \).

Exceptions: \( \text{SuperType} \) is already a supertype of \( T \); \( \text{SuperType} \) is a subtype of \( T \).

- procedure \texttt{Unlink\_From\_Supertype} ( \( T \) : in out Type; \( \text{SuperType} \) : in out Type );

\[
\begin{array}{l}
\text{Unlink\_From\_Supertype} \\
\text{Object\_Management\_Framework} \\
T? : \text{Types} \\
\text{SuperType}? : \text{Types} \\
(T?, \text{SuperType}?) \in \text{Subtype} \\
\text{Subtype}' = \text{Subtype} \setminus \{(T?, \text{SuperType}?)\} \\
T? \in \text{dom Subtype}'
\end{array}
\]

Remove \( \text{SuperType} \) from \( \text{Supertypes\_Of} ( T ) \).

Exceptions: \( \text{SuperType} \) is not in \( \text{Supertypes\_Of} ( T ) \); \( \text{SuperType} \) is the last remaining supertype of \( T \), so unlinking it would detach \( T \) from the type hierarchy.
• function Supertypes_Of ( T : in Type ) return set of Type;

```
function Supertypes_Of ( T : in Type ) return set of Type;
```

\[
\Xi \text{Object\_Management\_Framework}
\]

\[
T? : \text{Types}
\]

\[
\text{Supertypes! : } \mathbb{P} \text{ Types}
\]

\[
T? \in \text{Entities}
\]

\[
\text{Supertypes!} = \text{Subtype}[\{T?\}]
\]

Returns all of T’s (immediate) supertypes. Note that T is always guaranteed to have at least one supertype, by the semantics of Destroy and Unlink\_From\_Supertype.

• function Subtypes_Of ( T : in Type ) return set of Type;

```
function Subtypes_Of ( T : in Type ) return set of Type;
```

\[
\Xi \text{Object\_Management\_Framework}
\]

\[
T? : \text{Types}
\]

\[
\text{Subtypes! : } \mathbb{P} \text{ Types}
\]

\[
T? \in \text{Entities}
\]

\[
\text{Subtypes!} = \text{Subtype}^{-1}[\{T?\}]
\]

Returns all of T’s (immediate) subtypes. This function returns the empty set if T has no subtypes.

• procedure Add\_Attribute ( T : in out Type;

```
procedure Add\_Attribute ( T : in out Type;
```

\[
A : \text{in Attribute }
\]

\[
\Delta \text{Object\_Management\_Framework}
\]

\[
T? : \text{Types}
\]

\[
A? : \text{Attributes}
\]

\[
T? \in \text{Entities}
\]

\[
A? \in \text{Attributes}
\]

\[
(T?, A?) \not\in \text{Object\_Attributes}
\]

\[
\text{Object\_Attributes}' = \text{Object\_Attributes} \cup
\]

\[
\{ T : \text{Subtype}^{-1}[\{T?\}] \quad \bullet \quad (T, A?) \}\cup
\]

\[
\{(T?, A?)\}
\]

Adds A to T’s set of attributes. A type cannot have the same attribute twice, nor two attributes with the same name and type.
Exceptions: A is already an attribute of T; T already has an attribute whose name and type are the same as that of A.

- procedure Remove_Attribute ( T : in out Type;
  A : in Attribute );

\[
\begin{align*}
\text{Remove\_Attribute} & \\
\Delta & \text{Object\_Management\_Framework} \\
T? & : \text{Types} \\
A? & : \text{Attributes} \\
(T?, A?) & \in \text{Subtype}^{+} \{ \{ T? \} \} \subseteq \text{Object\_Attributes} \\
\text{Object\_Attributes}' & = \text{Object\_Attributes} \backslash \\
\{ T : \text{Subtype}^{-1} \{ \{ T? \} \} \} \backslash \\
\{ (T?, A?) \}
\end{align*}
\]

Removes A from T's set of attributes.

Exceptions: A is not an attribute of T; attempt to remove an attribute that is inherited from a supertype.

- function Attributes_Of ( T : in Type ) return set of Attribute;

\[
\begin{align*}
\text{Attributes\_Of} & \\
\exists & \text{Object\_Management\_Framework} \\
T? & : \text{Types} \\
A! & : \mathbb{P} \text{ Attributes} \\
T? & \in \text{Entities} \\
A! & = \text{Object\_Attributes} \{ \{ T? \} \} \backslash \\
\text{Object\_Attributes} \{ \text{Subtype}^{+} \{ \{ T? \} \} \}
\end{align*}
\]

Returns all of T's attributes. The empty set is returned if T has no attributes.

- procedure Add_OPERATION ( T : in out Type;
  O : in Operation );
_Add_Operation_

\[
\text{Add\_Operation}\quad \triangle \text{Object\_Management\_Framework}
\]
\[
T? : \text{Types}
\]
\[
O? : \text{Operations}
\]
\[
T? \in \text{Entities}
\]
\[
O? \in \text{Entities}
\]
\[
(T?, O?) \not\in \text{Entity\_Operations} \land
\]
\[
(\exists A : \text{Object\_Attributes} \mid \{ T? \} \bullet \text{Attribute\_Value}(A) = O?)
\]
\[
\text{Entity\_Operations}^t = \text{Entity\_Operations} \cup
\]
\[
\{ T : \text{Subtype}^{-1} \mid \{ T? \} \bullet (T, O?) \}\cup
\]
\[
\{(T?, O?)\}
\]
\[
\text{Object\_Attributes}^t = \text{Object\_Attributes} \cup
\]
\[
\{ T : \text{Subtype}^{-1} \mid \{ T? \} \cup \{ T? \}; A : \text{Attributes} \mid
\]
\[
\text{Attribute\_Name}(A) = \text{Operation\_Name}(O?) \land
\]
\[
\text{Attribute\_Type}(A) = \text{Entity\_Type}(O?) \land
\]
\[
\text{Attribute\_Value}(A) = O? \bullet (T, A)\}
\]

Adds O to T's set of operations. A type cannot have the same operation added

twice, and a type cannot have two operations with the same name and signature.

Exceptions: O is already an operation of T; another operation with the same

name and signature as O is already an operation of T.

This operation is provided as a convenient shorthand for adding an attribute to

a type whose name and type are that of O, and whose value is O (i.e., adding

an operation to a type is the same as adding an attribute to the type whose

value is the operation). This property is specified by the last predicate.

- procedure Remove\_Operation ( T : in out Type;

O : in Operation );
Removes O from T's set of operations.

Exceptions: O is not an operation of T; attempt to remove an operation inherited from a supertype.

- function Operations_Of ( T : in Type ) return set of Operation;

\[
\text{Operations\_Of} \\
\text{Object\_Management\_Framework} \\
T? : Types \\
O? : Operations \\
(T?, O?) \in \text{Subtype}^+ ( \{ T? \} \setminus \{ \text{Entity\_Operations} \} \setminus \{ T : \text{Subtype}^{-1} \setminus \{ T? \} \} \cup \{ T, O? \} \setminus \{ (T?, O?) \} \\
\{ \text{Entity\_Attributes}\setminus = \text{Object\_Attributes} \setminus \{ T : \text{Subtype}^{-1} \setminus \{ T? \} \setminus \{ T\} \cup \{ T? \}; \ A : \text{Attributes} | \text{Attribute\_Name}(A) = \text{Operation\_Name}(O?) \wedge \text{Attribute\_Type}(A) = \text{Entity\_Type}(O?) \wedge \text{Attribute\_Value}(A) = O? \bullet (T, A) \}
\]

Returns all of T's operations. The empty set is returned if T has no operations.

- procedure Add\_Constraint ( T : in out Type; 
  C : in Constraint );
\_Add\_Constraint

\Delta \text{Object\_Management\_Framework}
\[ T? : \text{Types} \]
\[ C? : \text{Constraints} \]

\[ T? \in \text{Entities} \]
\[ C? \in \text{Entities} \]
\[ (T?, C?) \not\in \text{Entity\_Constraints} \wedge \]
\[ \neg (\exists A : \text{Object\_Attributes} \mid \{T?\} \mid \bullet \]
\[ \text{Attribute\_Value}(A) = C? \]
\[ \text{Entity\_Constraints}' = \text{Entity\_Constraints} \cup \]
\[ \{T : \text{Subtype}\^1+ \mid \{T?\} \mid \bullet \}
\[ (T, C?)\} \cup \]
\[ \{(T?, C?)\} \]
\[ \text{Object\_Attributes}' = \text{Object\_Attributes} \cup \]
\[ \{T : \text{Subtype}\^1+ \mid \{T?\} \mid \cup \{T?\}; A : \text{Attributes} \mid \]
\[ \text{Attribute\_Name}(A) = \text{Operation\_Name}(C?) \wedge \]
\[ \text{Attribute\_Type}(A) = \text{Entity\_Type}(C?) \wedge \]
\[ \text{Attribute\_Value}(A) = C? \bullet (T, A)\}

Adds C to T’s set of constraints. A type cannot have the same constraint added twice, and a type cannot have two constraints with the same name and signature. Note that \textbf{Add\_Constraint} simply adds C to the set of constraints defined on a particular type—it does not enforce the constraint. See the definition of type \textbf{Constraint} for the semantics of constraint enforcement.

Exceptions: C is already a constraint of T; another constraint with the same name and signature as C is already a constraint of T.

This operation is provided as a convenient shorthand for adding an attribute to a type whose name and type are that of C, and whose value is C (i.e., adding a constraint to a type is the same as adding an attribute to the type whose value is the constraint). This property is specified by the last predicate.

- \textbf{procedure Remove\_Constraint} ( T : in out Type;
  C : in Constraint );
Remove_Constraint

\[ \text{Remove_Constraint} \]
\[ \Delta \text{Object\_Management\_Framework} \]
\[ T? : \text{Types} \]
\[ C? : \text{Constraints} \]

\[(T?, C?) \in \text{Subtype}^+ \{ \{ T? \} \} \subseteq \text{Entity\_Constraints} \]
\[ \text{Entity\_Constraints}' = \text{Entity\_Constraints}\setminus \{ T : \text{Subtype}^{-1}\{ \{ T? \} \} \bullet (T, C?) \} \]
\[ \{ (T?, C?) \} \]

\[ \text{Object\_Attributes}' = \text{Object\_Attributes}\setminus \{ T : \text{Subtype}^{-1}\{ \{ T? \} \} \cup \{ T? \} ; A : \text{Attributes} | \]
\[ \text{Attribute\_Name}(A) = \text{Operation\_Name}(C?) \land \]
\[ \text{Attribute\_Type}(A) = \text{Entity\_Type}(C?) \land \]
\[ \text{Attribute\_Value}(A) = C? \bullet (T, A) \}

Removes C from T’s set of constraints.

Exceptions: C is not a constraint of T.

- function \text{Constraints\_Of} ( T : \text{in Type} ) return set of Constraint;

\[ \text{Constraints\_Of} \]
\[ \Xi \text{Object\_Management\_Framework} \]
\[ T? : \text{Types} \]
\[ C! : \text{P \text{Constraints}} \]

\[ T? \in \text{Entities} \]
\[ C! = \text{ran} ( \{ T? \} < \text{Entity\_Constraints} \geq \]
\[ \text{Entity\_Constraints} \uplus \text{Subtype}^+ \{ \{ T? \} \} \downarrow \}

Returns all of the constraints that can apply to T. The empty set is returned if T has no applicable constraints.

- function \text{Enforced\_Constraints\_Of} ( T : \text{in Type} ) return set of Constraint;

\[ \text{Enforced\_Constraints\_Of} \]
\[ \Xi \text{Object\_Management\_Framework} \]
\[ T? : \text{Types} \]
\[ C! : \text{P \text{Constraints}} \]

\[ T? \in \text{Entities} \]
\[ C! = \text{Enforced\_Constraints} \downarrow \{ T? \} \uparrow \]
Returns all of the constraints on \( T \) that are currently enforced. The empty set is returned if \( T \) has no enforced constraints.

- function Extent_Of ( \( T : in \) Type ) return set of MetaType;

\[
\text{Extent}_{Of} \quad \Xi \text{Object\_Management\_Framework} \\
T? : \text{Types} \\
E! : \mathbb{P} \text{ENTITY} \\
\]

\[
T? \in \text{Entities} \\
E! = \text{Entity\_Type}^{-1} \cup \{ T? \}
\]

Returns the set of all instances of type \( T \). The empty set is returned if \( T \) has no instances.

**Attribute:** Type Attribute has one supertype: Object. It provides following operations:

- function Create ( \( N : in \) String; \\
  \( T : in \) MetaType ) return Attribute;

\[
\text{Create\_Attribute} \\
\text{Create\_Object\_Types\_Shared} \\
\text{Attr\_Name}? : \text{ATTRIBUTE\_NAMES} \\
\text{Attr\_Type}? : \text{Types} \\
\]

\[
\text{New\_Object!} \in \text{Attributes} \\
\text{Attribute\_Name'} = \text{Attribute\_Name} \cup \\
\{ \text{New\_Object!} \mapsto \text{Attr\_Name}? \} \\
\text{Attribute\_Type'} = \text{Attribute\_Type} \cup \\
\{ \text{New\_Object!} \mapsto \text{Attr\_Type}? \} \\
\text{Entity\_Type'} = \text{Entity\_Type} \cup \\
\{ \text{New\_Object!} \mapsto \text{Attribute} \}
\]

Creates a new attribute, named \( N \), whose value is of type \( T \).

- procedure Destroy ( \( A : in \) out Attribute );
Destroys a given attribute and removes the attribute from any type that has it.

- procedure Set_Value ( A : in out Attribute;
  V : in MetaType );

  Sets the value of a given attribute to V.

  Exceptions: The type of V is not the same as the type of A.

- function Get_Value ( A : in Attribute ) return MetaType;

  Returns the current value of a given attribute.

  Exceptions: Attribute has no value.

- function Attribute_Name ( A : in Attribute ) return String;
Get_Attribute_Name

\[ A? : \text{Attributes} \]
\[ N! : \text{ATTRIBUTE\_NAMES} \]
\[ A? \in \text{Entities} \]
\[ N! = \text{Attribute\_Name}(A?) \]

Returns the name of an attribute.

- function Attribute\_Type ( A : in Attribute ) return MetaType;

Get_Attribute\_Type

\[ A? : \text{Attributes} \]
\[ T! : \text{Types} \]
\[ A? \in \text{Entities} \]
\[ T! = \text{Attribute\_Type}(A?) \]

Returns the type of an attribute.

**Operation:** Type Operation has one supertype: Object. It provides the following operations:

- function Create ( Name : in String;
  Arguments : list of Parameter;
  Return\_Type : MetaType )
  return Operation;

Create\_Operation

Create\_Object\_Types\_Shared
\[ Name? : \text{OPERATION\_NAMES} \]
\[ Arguments? : \text{Signature} \]
\[ New\_Object! \in \text{Operations} \]
\[ Operation\_Name' = Operation\_Name \cup \{ \text{New\_Object!} \mapsto \text{Name}? \} \]
\[ Operation\_Signature' = Operation\_Signature \cup \{ \text{New\_Object!} \mapsto \text{Arguments}? \} \]
\[ Entity\_Type' = Entity\_Type \cup \{ \text{New\_Object!} \mapsto \text{Operation} \} \]
Creates a new operation with a given name and formal parameter list. Arguments is the empty list for operations with no parameters.

- procedure Destroy ( O : in out Operation );

```
Destroy_Operation
Destroy_Object_Types_Shared
O? ∈ Operations ∪ Entities
Operation_Name' = \{ O? \} ⊆ Operation_Name
Operation_Signature' = \{ O? \} ⊆ Operation_Signature
Entity_Operations' = Entity_Operations ⊃ \{ O? \}
```

Destroys an operation and removes it from any type of which it is an operation.

- procedure Set_Arguments ( O : Operation;
  Arguments : list of Parameter );

```
Set_Arguments
\Delta Object_Management_Framework
O? : Operations
Arguments? : seq Parameter
Current_Signature : Signature

O? ∈ dom Operation_Signature
Current_Signature = Operation_Signature(O?)
Current_Signature.Parameter_List = Arguments?
Operation_Signature' = Operation_Signature⊕
{ O? ↦ Current_Signature }
```

Changes an operation's formal parameter list.

- function Get_Arguments ( O : Operation ) return list of Parameter;

```
Get_Arguments
\Xi Object_Management_Framework
O? : Operations
A! : seq Parameter
Current_Signature : Signature

O? ∈ dom Operation_Signature
Current_Signature = Operation_Signature(O?)
A! = Current_Signature.Parameter_List
```

Returns a given operation's formal parameter list.
• procedure Set_Return_Type ( O : Operation;
    Return_Type : MetaType );

    Set_Return_Type
    δObject_Management_Framework
    O? : Operations
    New_Return_Type? : Types
    Current_Signature : Signature

    O? ∈ dom Operation_Signature
    Current_Signature = Operation_Signature(O?)
    Current_Signature. Return_Type = New_Return_Type?
    Operation_Signature' = Operation_Signature⊕
    { O? → Current_Signature }

    Changes an operation’s return type.

• function Get_Return_Type ( O : Operation ) return MetaType;

    Get_Return_Type
    ≡Object_Management_Framework
    O? : Operations
    R! : Types
    Current_Signature : Signature

    O? ∈ dom Operation_Signature
    Current_Signature = Operation_Signature(O?)
    R! = Current_Signature. Return_Type

    Returns a given operation’s return type.

• procedure Invoke ( O : Operation;
    Arguments : list of MetaType );

    Executes a given operation with a given actual parameter list.

    Exceptions: Argument list does not contain the same number or types of pa-
    rameters as those specified in the operation’s definition.

• procedure Return ( O : Operation );

    Causes an executing operation to terminate normally.

    Exceptions: Operation not executing.
• procedure Return_Abnormally ( O : Operation;
        E : Exception );

    Causes an executing operation to terminate abnormally and raise a specified exception.

    Exceptions: Operation not executing.

3.1.3 Satisfying the Cross-Cutting Requirements

Completeness: By definition, the type model satisfies the type and computational completeness requirement.

First-class status and identity: All entities in the type model have first-class status. Type Object is defined to satisfy the identity requirement, as do all types that inherit from it.

Meta-data: Many operations are defined on type Type to provide access to the definition of a type or instance—e.g., Supertypes_Of, Subtypes_Of, Operations_Of, etc. Further, the use of a Smalltalk-like metatype model ensures that types are treated as any other objects, so they, too, satisfy the meta-data requirement.

Dynamic control: The operations defined on these types provide pervasive dynamic control over the definition of both the types and their instances.

Generality and heterogeneity: The fully extensible type model, combined with ubiquitous first-class status, identity, and type and computational completeness, permits the definition of many kinds of objects and object semantics.
3.2 Persistence

3.2.1 Overview

A persistence model that supports complex applications should have the following characteristics:

**Completeness:** It should allow any instance of any type to persist for as long as necessary. That is, every object can be thought of as having a special persistence attribute, and an object may be designated as persistent or transient based on the value of this attribute. This is modeled in the Z specification for type Object_Management_Framework by the inclusion of the predicate that mandates the presence of a persistence attribute for all instances of type Object.

Note that the persistence attribute is really a meta-attribute, because all objects have the attribute as an intrinsic property. Thus, applications cannot manipulate a persistence attribute in all the same ways that they can manipulate other types of attributes. For example, while they may be able to set and retrieve the value of an object’s persistence attribute, they may not destroy an object’s persistence attribute. This property is enforced in the Z schema specification for type Object_Management_Hierarchy by the predicate that mandates the existence of a persistence attribute for all objects.

**First-class status and identity:** It should support the definition and manipulation of an object’s persistence attribute independently of any other attribute (i.e., persistence should be orthogonal to any other property of an object [7]). Thus, the persistence attribute should be a first-class attribute. Further, it should be possible for different objects to share the same persistence attribute. This provides, for example, a straightforward means of defining reachability-based persistence, in which all objects that are navigationally reachable (transitively) from a persistent object also become
persistent—i.e., reachable objects can be said to share the same persistence attribute, and thus, all reachable objects become persistent at the same time. The ability for objects to share a persistence attribute suggests that the persistence attribute itself should have identity. This is modeled in the definition of the Z schema specification for type `Object_Management_Framework` by defining the persistence attribute as an instance of type `Attribute`, which is itself a subtype of type `Object`.

We note that two interpretations of the first-class status and identity requirements on the persistence attribute may be given: that the attribute is an object containing a value, where the attribute object is a first-class object with identity, and that the attribute contains an value that is a first-class object with identity. In the first case, the persistence attribute object may be shared, and in the second case, the value of the persistence attribute may be shared. Each of these models provides different semantics that are useful for different applications, and thus, both are supported in the object management framework—i.e., persistence attributes themselves are first-class objects with identity, and the values they hold may be first-class objects with identity.

**Dynamic control:** It should permit an application to decide dynamically which objects will persist. Further, it should allow an application to indicate dynamically what the *extent* of persistence is. The extent of persistence is defined as the set of objects that become persistent when a given object does. Similarly, the persistence model should allow an application to force the destruction of a persistent object dynamically, when it believes that a persistent object no longer needs to persist (i.e., that the object has become useless or meaningless).

**Meta-data:** It should allow an application to find out whether or not a given object is persistent at any time.
**Generality and heterogeneity:** It should readily support the definition of multiple persistence models. For example, some commonly used persistence models include *persistence-by-type* (e.g., [109, 29]), where all instances of a particular type persist, *persistence-by-instance* (e.g., [6, 124]), where objects that satisfy some predicate become persistent, and reachability-based persistence.

### 3.2.2 Formalization

To satisfy the cross-cutting requirements, we add the following attribute and operations to the declaration of type `Object`:

- **Persistent : Boolean;**

  Declares `Persistent` to be an attribute of type `Object` type. The value of this attribute indicates whether or not an object is persistent. Note that, since `Persistent` is specified as an attribute, it is actually defined to be an instance of type `Attribute`, and as such, it is a first-class entity with identity.

  Note that the `Persistent` attribute cannot be deleted from any subtypes of `Object`, because it is illegal to remove an attribute from a subtype that it inherits from a supertype. The only way to remove the `Persistent` attribute is to remove it from type `Object`. The ability to remove this attribute may be useful in systems that are not persistence-capable. In systems that are persistence-capable, the `Persistent` attribute should not be removable from type `Object`. This restriction may be enforced in any number of ways (e.g., through access control or through the implementation of operation `Remove_Attribute`), so we leave the manner of enforcement as an implementation-specific detail.

- **function Is_Persistent ( O : Object ) return Boolean;**
Returns the current value of an object’s Persistent attribute.

- **procedure Make_Persistent ( O : Object );**

```
Make_Persistent
\delta Object_Management_Framework
O? : Objects

O? ∈ Entities
Attribute_Value' = Attribute_Value ⊕ {(O?, True)}
```

Sets the value of O’s Persistent attribute to True. This makes O persistent, along with any other objects whose persistence depends on O. For example, when using navigational reachability as the definition for persistence, making O persistent implicitly makes all objects reachable from O persistent as well.

The persistence of an object, O₁, may depend on that of another object, O₂, if the objects either share the same persistence attribute or if the value of O₁’s persistence attribute depends on the value of O₂’s persistence attribute (e.g., they may share the same attribute value, or O₁’s persistence attribute may be derived from the value of O₂’s persistence attribute).

- **procedure Make_Transient ( O : Object );**

```
Make_Transient
\delta Object_Management_Framework
O? : Objects

O? ∈ Entities
Attribute_Value' = Attribute_Value ⊕ {(O?, False)}
```
Sets the value of the \texttt{Persistent} attribute to \texttt{False}.

### 3.2.3 Satisfying the Cross-Cutting Requirements

**Completeness:** By adding the \texttt{Persistent} attribute to type \texttt{Object}, we ensure that all subtypes of objects are potentially persistent. Thus, the persistence capability can apply to all entities.

**First-class status and identity:** The \texttt{Persistent} attribute is modeled as a first-class entity with identity. We note that the modeling of persistence in this way has the beneficial effect of providing a very straightforward way to model extent of persistence—i.e., as the set of objects that share the same persistence attribute (since this ensures that a change of the persistence attribute’s value affects the persistence status of all the objects that share the attribute).

**Meta-Data:** We supplement the existing meta-data with the operation \texttt{IsPersistent}.

**Dynamic control:** The operations \texttt{MakePersistent} and \texttt{MakeTransient} provide dynamic control over which objects are persistent. Further, the first-class status and identity of the \texttt{Persistent} attribute means that the extent of persistence for any object could potentially be changed dynamically (i.e., because the value of an object’s \texttt{Persistent} attribute can be changed dynamically).

**Generality and heterogeneity:** The modeling of persistence defined by the object management framework supports, by default, instance-based persistence, and it readily accommodates alternative persistence models and implementations. For example, persistence-by-type can be modeled by defining the persistence attribute as a constant:

\begin{verbatim}
Persistent : constant Boolean := True;  -- Or False
\end{verbatim}
Creation-time persistence decisions (as provided, for example, in E [89] and ObjectStore [79]) are modeled by adding a parameter to the Create operation:

\[
\text{function Create ( Is\_Persistent : Boolean ) return Object;}
\]

Note that this operation is simply a subtype of the Create operation defined on type Object. Further, the operational specification of Make\_Persistent and Make\_Transient readily facilitates alternative implementations and semantics.

### 3.3 Navigational and Associative Access

#### 3.3.1 Overview

**Completeness:** It should be possible to navigate from an object to any subcomponent (represented by the object’s set of attributes). Similarly, it should be possible to formulate queries over arbitrary collections of objects. Completeness requires that it be possible to express any necessary query—that is, associative access must be computationally complete.

**First-class status and identity:** The connection between an object and a subcomponent over which navigation occurs (i.e., an edge) should itself be a first-class object with identity. This requirement is particularly important in software engineering applications, where it may be necessary to annotate an edge with edge-specific information, such as branching conditions. In addition, queries should also be modeled as a first-class objects with identity.

**Dynamic control:** It should be possible for an application to select or define a navigational path or a query dynamically. It should also be possible for an application to freely intermix navigation and querying throughout its execution.

**Meta-data:** Meta-data support should include sufficient information for an application to determine what navigational or associative access patterns can be used on
a particular object or set of objects. For example, it should be possible to determine dynamically how to navigate through a particular object, which queries are available, and what the definitions of those queries are.

**Generality and heterogeneity:** It should be possible to define any meaningful navigational access path and query.

### 3.3.2 Formalization

The cross-cutting requirements are satisfied by the definitions provided in Chapter 3.1.2.2. Note in particular that because attributes are defined to be first-class objects, the connections between objects and their subcomponents are first-class objects. Similarly, queries are first-class objects because they are modeled as operations, which are first-class objects.

### 3.4 Consistency Management

#### 3.4.1 Overview

**Completeness:** It should be possible to state arbitrary consistency conditions and appropriate responses to violations. That is, the definition of consistency conditions and violation responses should be computationally complete. Further, the completeness requirement suggests that it should be possible to enforce consistency definitions over instances of any type of object.

**First-class status and identity:** Consistency definitions and violation responses should be modeled as first-class objects with identity.

**Dynamic control:** Dynamic control over several aspects of a consistency management mechanism should be present. First, it should be possible to enforce or relax any consistency condition on any object (or collection of objects) dynamically. Second, it
should be possible to define or modify consistency definitions and violation responses dynamically (subject to any constraints that must be imposed to preclude dangerous or undesirable behaviors). Finally, it should be possible to associate different violation responses with a particular consistency condition and object(s) dynamically; it should also be possible for the violation response to a particular consistency condition to differ from one object to another.

**Meta-data:** It should be possible to determine which consistency definitions are enforced on a given object at any time; how consistency violations are detected; and which consistency violation responses are used.

**Generality and heterogeneity:** In terms of supporting generality and heterogeneity, a consistency management mechanism should permit the definition of alternative consistency conditions, allow different kinds of violations to be identified differently (e.g., eagerly or lazily), and support multiple violation response semantics (e.g., preclusion, roll-forward, and roll-backward).

### 3.4.2 Formalization

To satisfy the cross-cutting requirements, we define types *Constraint* and *Action* as follows:

**Constraint:** Type *Constraint* has one supertype: *Operation*. It has the following operations:

- procedure Enforce ( C : Constraint;
                  On_Objects : set of MetaType;
                  Enforcement_Info : set of <Operation, When, Action> );

where *When* = {Preinvoke, Precondition, Postcondition, Postexecution}.
Enforces a constraint on one or more objects. **Enforcement_Info** represents information that indicates how the constraint should be enforced: the operations in which the constraint should be checked, at what point the constraint should be checked (before the operation is invoked, as a precondition to the operation, as a postcondition to the operation, or after the invocation of the operation), and an optional action to be taken upon (attempted) violation of the constraint. If no actions are specified, the default action is to raise an exception upon (attempted) violation of the constraint.

The specification of this operation ensures that it is possible for applications to control precisely, and on an instance-by-instance basis (if necessary), all aspects of consistency management. An application can specify that a one set of actions be taken in response to violation of a constraint that is enforced on one object, and that a different set of actions be taken in response to a violation of the same constraint on another object. Similarly, applications can indicate when a constraint should be checked, which allows them to control the model of consistency enforcement used (e.g., violation preclusion, rollback semantics, or substitution, in which a different operation is invoked in place of the one that might have violated the constraint).

Note that all instances of type **Constraint** must be operations that return an indication of whether the property they check is satisfied. This value must be
in the set \{True, False, Partial, Unknown\}. In some cases, constraint satisfaction cannot be determined [107] (e.g., because of an inability to obtain access to some of the objects affected by the constraint); when this is the case, Unknown should be returned. The return value Partial is provided as a hook for implementations in which partial satisfaction of a constraint (e.g., if a system compiles but does not link, the system partially satisfies an “is buildable” constraint) can be determined meaningfully. The ways in which this determination occurs are implementation-specific. It could, for example, be as simple as allowing individual constraint implementations check for partial satisfaction, or it could be more complicated, such as defining a subtype of Constraint that includes operations that allow constraints to be defined in terms of “subconstraints” (e.g., conjunctions or disjunctions of constraints) and returning Partial if any subconstraints are satisfied.

Exceptions: the types of the constraint’s parameters are not compatible with those of any operation defined on the object; the types of one of the action’s parameters are not compatible with those of the constraint; one of the operations in which the constraint was to be checked is not an operation applicable to On_Objects.

- procedure Relax ( C : Constraint;
  On_Object : MetaType;
  Enforcement_Info : set of <Operation, When, Action> );

where When = \{Preinvoke, Precondition, Postcondition, Postexecution\}.
Relax

```
\Delta \text{Object\_Management\_Framework}
C? : \text{Constraints}
On? : \text{ENTITY}
Info? : \mathbb{P} \text{Enforcement\_Info}
```

```
C? \in \text{Entities}
On? \in \text{Entities}
\forall I : \text{Info}? \bullet ( (C?, On?), I) \in \text{Enforced\_Constraint\_Info}
\text{Enforced\_Constraint\_Info}' = \text{Enforced\_Constraint\_Info} \setminus \{ I : \text{Info}? \bullet ( (C?, On?), I) \}
```

Relaxes an enforced constraint on an object. If the set of \text{Enforcement\_Info} is empty, all enforcements of the constraint on the given object are relaxed.

Exceptions: Constraint not currently enforced on object; the enforcement information specified is not valid (i.e., the constraint was not enforced on On\_Object with the specified \text{Enforcement\_Info}).

- function \text{Is\_Enforced} ( C : \text{Constraint}; On\_Object : \text{MetaType})
  return Boolean;

```
\exists \text{Object\_Management\_Framework}
C? : \text{Constraints}
On? : \text{ENTITY}
\text{Enforced!} : \text{Types}
```

```
C? \in \text{Entities}
On? \in \text{Entities}
((C?, On?) \in \text{dom Enforced\_Constraint\_Info} \land 
\text{Enforced!} = \text{True}) \lor \text{Enforced!} = \text{False}
```

Determines whether or not a constraint is currently enforced on a given object.

- function \text{Enforced\_On} ( C : \text{Constraint}) return set of \text{MetaType};
Indicates on which objects a given constraint is currently enforced. The empty set is returned if the constraint is not currently enforced on any objects.

- function Get_Enforcement_Info ( C : Constraint;
  On_Object : MetaType )
  return set of <Operation, When, Action>;

where \( \text{When} = \{ \text{Preinvoke, Precondition, Postcondition, Postexecution} \} \).

\[
\begin{align*}
\text{Get}_\text{Enforcement}_\text{Info} & \text{Object}_\text{Management}_\text{Framework} \\
C? & : \text{Constraints} \\
\text{On}? & : \text{ENTITY} \\
\text{Info!} & : \text{Enforcement}_\text{Info} \\
C? & \in \text{Entities} \\
\text{On}? & \in \text{Entities} \\
\text{Info!} & = \text{Enforced}_\text{Constraint}_\text{Info}\{ (C?, \text{On}?) \} \\
\end{align*}
\]

Returns information describing when a given constraint, which is enforced on a specified object, will be checked and what actions will be taken in response to (attempted) violations.

Exceptions: Constraint not enforced on object.

- function Satisfied ( C : Constraint;
  On_Objects : set of MetaType )
  return \{ True, False, Partial, Unknown \};

Determines whether or not a constraint, which is enforced on a given set of objects, is currently satisfied. Note that this operation does not cause any actions associated with the constraint to be executed—it simply tests for the satisfaction of the constraint. Note that because the test for satisfaction of a constraint depends on the definition of the constraint, no \(Z\) specification is given.

**Action:** Type **Action** has one supertype: **Operation**. For the most part, **Actions** are identical to the **Operation** ADT definition. They define only a few additional operations:
• function Applies_To ( A : Action ) return set of MetaType;

\[
\text{Applies}_\text{To}
\]

\[\Xi \text{Object}_\text{Management}_\text{Framework}
A? \colon \text{Actions}
E! : \mathbb{P} \text{ENTITY}
\]

\[
A? \in \text{Entities}
E! = \{ \text{Info} : \text{ran Enforced}_\text{Constraint}_\text{Info}; E : \text{ENTITY};
C : \text{Constraints} |
\text{Info.Action}_\text{To}_\text{Invoke} = A? \land
(E, C) \in \text{Enforced}_\text{Constraint}_\text{Info}^{-1} \| \{ \text{Info} \} \| \bullet E\}
\]

Determines the set of objects for which a given action will be taken upon violation of some constraint(s). The empty set is returned if the action is not currently associated with any object.

• function Action_For ( A : Action;
Obj : MetaType ) return set of Constraint;

\[
\text{Action}_\text{For}
\]

\[\Xi \text{Object}_\text{Management}_\text{Framework}
A? : \text{Actions}
E? : \text{ENTITY}
C! : \mathbb{P} \text{ENTITY}
\]

\[
A? \in \text{Entities}
E? \in \text{Entities}
C! = \{ \text{Info} : \text{Enforcement}_\text{Info}; C : \text{Constraints} |
(E?, C) \in \text{dom Enforced}_\text{Constraint}_\text{Info} \land
\text{Info} \in \text{Enforced}_\text{Constraint}_\text{Info} \| \{ (E?, C) \} \| \land
\text{Info.Action}_\text{To}_\text{Invoke} = A? \bullet C\}
\]

Given an object for which A is an action, this operation determines for which of Obj’s enforced constraints A is an action to be taken upon violation.

Exceptions: A is not an action associated with any constraint enforced on Obj.

3.4.3 Satisfying the Cross-Cutting Requirements

Completeness: As noted in Chapter 3.1.3, the type model is computationally complete. Thus, the definition of constraints and actions as operations means that both
kinds of entities are computationally complete. Further, the definition of type Constraint permits constraints to be enforced over arbitrary types of objects.

**First-class status and identity:** Since constraints and actions are modeled as operations, and operations are first-class objects, constraints and actions are first-class objects.

**Meta-Data:** The operations defined on types Object, Constraint, and Action permit applications to determine, at any time, which constraints are associated with a particular object, whether or not the constraints are satisfied, what actions are taken upon violation, and how enforcement occurs.

**Dynamic control:** Full dynamic control over the enforcement and relaxation of constraints is provided as part of the definition of constraints.

**Generality and heterogeneity:** The framework facilitates the definition of a wide variety of alternative consistency management semantics. It supports both instance- and type-level control over enforcement of constraints (because types are first-class objects). Eager and lazy detection of violations and the full range of violation response semantics can occur simply by controlling when a particular constraint is checked.

3.5 **Concurrency Control**

3.5.1 **Overview**

**Completeness:** It should be possible to have concurrent access to any type of object, and such access should be controlled, as appropriate, to allow correct manipulation of the objects.
For purposes of discourse, we refer subsequently to *transactions* as units of concurrency control. That is, transactions serve to delineate the periods of time during which concurrent access to objects can occur.

**First-class status and identity:** Transactions should be modeled as first-class objects with identity. If log-based recovery [16] is used, the log should also be a first-class object with identity. Further, the recovery manager [16] (i.e., the component that ensures failure atomicity) should be a first-class object with identity.

**Dynamic control:** It should be possible to change properties of a transaction dynamically. For example, it should be possible to restructure a transaction (e.g., [84]), to change an atomic transaction to be non-atomic, and to create dependencies among transactions [27].

**Meta-data:** It should be possible to determine what properties a given transaction possesses.

**Generality and heterogeneity:** In terms of generality and heterogeneity, it must be possible to describe and implement any useful concurrency control and recovery semantics. This does not mean, however, that all types of transactions must be either interoperable or even allowed.

### 3.5.2 Formalization

We introduce the type *Transaction* into the framework as the unit of concurrency control. It has one supertype, *Object*, and provides the following operations:

- **function Create ( With Dependencies : set of <Transaction, Dependency>; With Properties : set of Property; return Transaction;**

  Creates a new transaction. The transaction may optionally have a set of dependencies (as defined in [27]) on other transactions (e.g., an abort dependency,
so that the new transaction will abort if others do), and it may have a set of properties it satisfies (e.g., atomicity or serializability). The set of dependencies and properties supported may vary from instantiation to instantiation. The initial status of the new transaction is Inactive. (See the definition of function Status later in this section for details.)

- **function Dependencies_Of ( T : Transaction )**
  
  return set of <Transaction, Dependency>;

  Returns the set of transactions on which a given transaction is dependent, and indicates the kinds of dependencies that exist among the transactions. The empty set is returned if T has no dependencies.

- **procedure Add_Dependency ( To : in out Transaction;**
  
  D : <Transaction, Dependency> );

  Creates a new dependency between two transactions and adds it to Dependencies_of ( To ).

  Exceptions: Dependency already exists; To has already committed or aborted.

- **procedure Remove_Dependency ( From : in out Transaction;**
  
  D : in <Transaction, Dependency> );

  Removes an existing dependency between two transactions from Dependencies_of ( To ).

  Exceptions: This dependency does not exist (i.e., D is not in Dependencies_of ( To )); From has already committed or aborted.

- **function Properties_of ( T : Transaction )**
  
  return set of Property;

  Returns the set of properties a given transaction satisfies. The empty set is returned if T has no specified properties.

- **procedure Add_Property ( To : in out Transaction;**
  
  P : in Property );
Adds a new property to the set of properties that a given transaction must satisfy. Note that the transaction may not satisfy the given property. If it does not, the property cannot be added to it.

Exceptions: Property already set; property not satisfied.

- **procedure Remove_Property** ( From : in out Transaction;  
  P : in Property );

Removes a property that a transaction currently satisfies.

Exceptions: The transaction does not satisfy the given property.

- **procedure Begin** ( T : Transaction );

  Begins the execution of a given transaction. T’s status should be **Inactive**.

  Exceptions: T’s status is not **Inactive**.

- **procedure Restart** ( T : Transaction );

  Starts the execution of a given transaction from the beginning. T’s status should be **Running** or **Suspended**.

  Exceptions: T’s status is not **Running** or **Suspended**.

- **procedure Suspend** ( T : Transaction );

  Temporarily suspends the execution of an ongoing transaction.

  Exceptions: Transaction not running.

- **procedure Resume** ( T : Transaction );

  Resumes the execution of a suspended transaction from the point at which it was suspended.

  Exceptions: Transaction not suspended.

- **procedure Commit** ( T : Transaction );

  Terminates the execution of a transaction normally. This causes its results to be committed in accordance with the set of properties the transaction satisfies. For example, if T is atomic, its results are committed in an all-or-nothing manner; if
it is not, some or all of its results may have already been committed. The commit of an operation must satisfy all of the transaction's dependencies. Thus, for example, the commit will not be allowed if $T$ is abort dependent on a transaction that already aborted.

Exceptions: Transaction aborted; transaction not running; dependencies violated; properties violated; unable to commit.

- **procedure** **Abort** ( $T$ : Transaction );

Terminates the execution of a transaction abnormally. Depending on the properties of the transaction, the transaction's results may be discarded in their entirety (i.e., if the transaction is atomic) or in part.

Exceptions: Transaction already committed; transaction not running.

- **procedure** **Status** ( $T$ : Transaction ) return **Status**;

where $Status = \{\text{Inactive, Running, Suspended, Committed, Aborted}\}$.

Indicates the current status of a transaction.

- **procedure** **Control**.Object ( $T$ : in Transaction; $O$ : in MetaType );

Indicates that a given object will be subject to concurrent access as part of a specified transaction. Access to the object will satisfy all $\text{Properties.0f} ( T )$ (e.g., if $T$ is serializable, $O$ will be accessed in a serializable fashion).

Exceptions: $O$ is already controlled by $T$; concurrent access conflict.

- **procedure** **Release**.Object ( $T$ : in Transaction; $O$ : in MetaType );

Indicates that an object, to which concurrent access is currently controlled by a given transaction, cannot be accessed subsequently as part of the transaction.

Exceptions: Access to $O$ is not controlled by $T$.

- **procedure** **Controlled**.By ( $O$ : in MetaType )

  return set of Transaction;
Determines in which transactions a particular object is being accessed. If the object is not being accessed as part of any transaction, the empty set is returned.

Table 3.1 summarizes the implications of the cross-cutting requirements on these areas of object management functionality.

3.6 Related Work

The Object Data Management Group (ODMG) has proposed a standard definition for primitive capabilities that all object-oriented databases must provide [8]. In this specification, the ODMG proposed a schema definition language (ODL), a query language (OQL), and standard language bindings for C++ and Smalltalk. This definition represents the nearest related work to the object management framework proposed in this chapter. The purpose of the ODMG standard specification is somewhat different from ours, however, which leads to a number of differences between that specification and our object management framework. In particular, the primary stated goal of the ODMG specification is “to put forward a set of standards allowing an ODBMS [object database management system] customer to write portable applications, i.e., applications that could run on more than one ODBMS product. The data schema, programming language binding, and data manipulation and query languages must be portable.” In essence, the goal of the ODMG’s standardization effort is to provide the object-oriented database equivalent of the relational database and SQL standards—to enforce enough common syntax and semantics across object-oriented database systems that clients could port their applications from one database system to another with as little pain as possible.\(^2\)

\(^2\)Different database systems may, of course, provide different “add-on” capabilities to appeal to different market segments, so application portability occurs only as long as a particular application uses only the common subset of functionality.
Table 3.1 Summary: Cross-Cutting Requirements as Constraints on Functional Requirements.

<table>
<thead>
<tr>
<th>High-Level Type Models</th>
<th>Persistence</th>
<th>Navigational Access</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Completeness</strong></td>
<td>Can apply any constructor to any type</td>
<td>Any instance of any type is potentially persistent</td>
</tr>
<tr>
<td><strong>First-Class Status</strong></td>
<td>Can refer to all objects uniformly (including types, operations)</td>
<td>An object’s persistence attribute is a first-class attribute</td>
</tr>
<tr>
<td><strong>Identity</strong></td>
<td>All objects have unique names distinct from their values</td>
<td>The persistence attribute has identity</td>
</tr>
<tr>
<td><strong>Dynamic Control</strong></td>
<td>Can dynamically define or modify types, instances</td>
<td>Can dynamically determine which objects persist (and extent of persistence)</td>
</tr>
<tr>
<td><strong>Meta Data</strong></td>
<td>Can dynamically determine object’s type (definition)</td>
<td>Can dynamically determine whether an object is persistent</td>
</tr>
<tr>
<td><strong>Generality/ Heterogeneity</strong></td>
<td>Readily supports the definition of different type and instance semantics</td>
<td>Readily supports multiple persistence models</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Associative Access</th>
<th>Consistency Management</th>
<th>Concurrency Control</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Completeness</strong></td>
<td>Can formulate queries over instances of any (collection) type; queries are computationally complete</td>
<td>Consistency definitions can be enforced over instances of any type of object; any consistency definition and violation response can be defined (computationally complete)</td>
</tr>
<tr>
<td><strong>First-Class Status</strong></td>
<td>Can refer to queries as first-class objects</td>
<td>Consistency definitions and violation responses are first-class objects; can be applied selectively to instances of types</td>
</tr>
<tr>
<td><strong>Identity</strong></td>
<td>Queries have names distinct from value/state</td>
<td>Consistency definitions and violation responses have names distinct from states</td>
</tr>
<tr>
<td><strong>Dynamic Control</strong></td>
<td>Can dynamically define or switch to associative access pattern</td>
<td>Can dynamically enforce and relax consistency definitions, change violation response, and specify new consistency definitions</td>
</tr>
<tr>
<td><strong>Meta Data</strong></td>
<td>Can dynamically obtain query definition</td>
<td>Can dynamically determine which consistency definitions and violation responses are enforced on an object</td>
</tr>
<tr>
<td><strong>Generality/ Heterogeneity</strong></td>
<td>Readily supports different kinds of queries</td>
<td>Readily supports multiple consistency definition and violation response semantics (e.g., preclusion, roll-forward, roll-backward)</td>
</tr>
</tbody>
</table>
The goal of providing application portability is a laudable one. Facilitating application portability, however, requires the specification of a wide spectrum of semantics—e.g., a complete type and data model, run-time semantics, etc.—to ensure that the common set of database capabilities are sufficient for defining an “average” application. With the enforcement of such a wide spectrum of semantics, bindings to particular languages becomes very difficult, due to impedance mismatches between the semantics of the database standard and a particular language. In fact, the mismatches are inevitable, given that no two languages define precisely the same semantics, and they are a significant reason why we did not choose to pursue an object-oriented database approach—either the language does not support features that the database does, or the database does not support the full range of the language’s semantics. Some of these mismatches are clear even in the bindings the ODMG produced for C++ and Smalltalk. For example, the C++ binding does not support several capabilities that are required by the standard, including type extents, key declarations, multiple names per object, referential integrity of unidirectional relationships, and return of multiple values from functions. Further, not all of the C++ standard types are required to be supported by the underlying database—in particular, support for unions, bit fields, and references may be omitted. This means that persistence and concurrency control need not apply to these types of objects.

Our goal is fairly different from that of the ODMG. Given that each programming language has different capabilities, we wanted to define an object management framework that describes classes of object management functionality that satisfies the orthogonality, flexibility, and ease-of-use requirements, as embodied by the set of cross-cutting requirements. We particularly did not want to impose a particular programming paradigm (not even object-oriented), or to overly constrain the set of semantic models or implementation strategies that could be used. Thus, with some
limitations, we would view the ODMG’s efforts as having approximated an instantiation of our framework for an object-oriented type model.

Several aspects of the current version of the ODMG’s proposed standard fail to satisfy the cross-cutting. Examples of these are:

- Attributes, relationships, and operations are not first-class objects.
- Queries and transactions apply only to persistent objects, which violates the completeness requirement.
- No dynamic control over the persistence decision. Objects must be declared persistent or transient at creation-time.
- Persistence cannot apply to transaction objects.
- References to persistent and transient objects are not the same. In particular, references to persistent objects are made using $\text{Ref}<\text{obj}>$, while transient objects are referenced using the host programming language’s native reference mechanism. This violates orthogonality. Further, the C++ binding requires a call to the method $\text{Persistent}\_\text{Object}::\text{mark}\_\text{modified}$ to indicate that the object has been modified, which serves to differentiate persistent and transient objects further.

We note that the ODMG’s specification does itself describe some of these as limitations, and it indicates an intent to address many of these limitations in the future. We take this as additional evidence in support of the definition of our object management framework.
CHAPTER 4

PLEIADES: A PROTOTYPE OBJECT MANAGEMENT SYSTEM FOR ADA

In the previous chapter, we applied the set of cross-cutting requirements proposed in Chapter 2.4 to the set of required object management capabilities discussed in Chapter 2.2 to produce an object management framework describing characteristics of categories of object management systems that we believe satisfy the orthogonality, flexibility, and ease-of-use goals. To help us evaluate this hypothesis, we produced a mapping of the object management framework to the Ada programming language. This activity produced a prototype database programming language, called Pleiades. In this chapter, we describe the functionality Pleiades provides. The following three chapters provide more detailed descriptions of some of the functional models and implementation strategies we employed in producing Pleiades. A detailed reference manual for Pleiades is available as well [112]. Much of the description of Pleiades in this chapter is taken from [113].

4.1 Language Features to Support Object Management

The Pleiades system has been developed to support our own efforts to build software environments and to explore the issues associated with object management. Although Pleiades is currently implemented as a set of extensions to Ada, the features it supports satisfy many of the requirements presented in Section 2.2 and are of general interest. In this section, we describe the major features of Pleiades. We also contrast these features with other approaches, and we discuss some current limitations of the system.
4.1.1 Appropriate High-Level Primitive Types

As noted in Section 2.2, graph, varying-length sequence, relation, and relationship types are pervasive in software engineering environments. To support the definition of these types, PLEIADES defines an extended set of type constructors that includes all of the “standard” programming language type constructors (e.g., record and array), plus constructs for describing graphs, varying-length sequences, relations, and relationships. In addition, PLEIADES defines a set of operations for creating and manipulating instances of these types just like any other built-in type, along with a set of exceptions that these operations may raise. A programmer need only describe a type, and PLEIADES automatically provides an abstract data type definition for that type.

The PLEIADES type model satisfies a number of the cross-cutting requirements. It supports the definition of, for example, graphs of relations, and relationships between graphs or relations. All types are first-class entities, and all instances of types have identity and are first-class entities. A set of predefined operations provides certain kinds of meta-data. In defining the semantics of each of the new type constructors, we have attempted to select a general model for each class of abstract data type, and thus, to facilitate the definition of higher-level models. We describe these type constructors and demonstrate their generality below.

4.1.1.1 Graphs

PLEIADES provides two type constructors to support graph type development: node and edge.

Nodes can have zero or more attributes, each of which can have any type. If the type of a node’s attribute is itself a node type, then the value of that attribute will be a reference to another node—that is, an implicit edge will exist. Operations to create and destroy nodes, to set and retrieve values of node attributes, and to dynamically
determine the type of a given node or any of its attributes are provided for all node types.

While the node constructor alone is sufficient to permit the definition of directed graph types, we have found that some applications require graphs that contain explicit edges; for example, some applications, such as the CFG builder, must annotate the edges of a graph with edge-specific information. Although it is possible to create a node that represents the edge, this is not as natural as incorporating an edge type. Thus, Pleiades supports the definition of explicit directed edge types as well as implicit ones. Edges, like nodes, can be attributed. Applications can examine explicit edges during graph traversals, or they can simply ignore explicit edges and traverse the graphs as though they contained implicit edges. An application that does not care about edges or their annotations therefore need not be aware of their existence.¹ Operations to create and destroy edges, to set and retrieve values of edge attributes, to determine dynamically the type of a given edge or any of its attributes, to obtain the source and target of an edge, and to traverse an edge are defined on all edge types.

Attributes in both nodes and edges can have computed values, which are derived dynamically from other values. For example, given the current date and a person’s birth date, the person’s age can be computed. Pleiades permits the values of computed attributes to be derived using any programmer-specified operation, and it allows the values of computed attributes to be derived either lazily (upon demand) or eagerly (whenever any of the data from which the attribute’s value is derived change), as the programmer chooses.

¹While the semantics of implicit edges are subsumed by those of explicit edges, we chose to provide the ability to define implicit edges for reasons of completeness; indeed, it may be more natural to use implicit edges for some graph types.
The Pleiades model of graphs is quite general. The node and edge abstractions can be used to define many different semantics for graphs, including directed, connected, and sets of nodes and edges.

4.1.1.2 Sequences

Pleiades introduces the sequence constructor to support the development of varying-length, ordered sequences of objects. Sequences are similar to arrays in that elements in a sequence can be accessed by their position within the sequence. They are also similar to linked lists in that inserting an element into a sequence causes all elements stored after the new element to be shifted; thus, inserting an element at position \( n \) causes the element that was formerly at that position to move to position \( n + 1 \). Sequences grow and shrink dynamically, so any number of elements may be inserted into a given sequence. Operations to create, destroy, insert into, remove from, retrieve from, and iterate over elements in a sequence are supported, as are operations to determine dynamically the types of a sequence and its elements. Sequences are type complete, so they can be defined over any type of object, including sequences, nodes, edges, relations, and relationships.

4.1.1.3 Relationships and Relations

Relationships are \( N \)-ary connections between entities. In Pleiades, the attributes of relationships can have any type, including graphs, relations, and relationships. The values of relationship fields can also be computed. Unlike traditional relational database models, instances of relationship types are first-class objects with identity. Operations are provided to create and destroy relationships, to set and retrieve values of their attributes, and to determine dynamically the type of a given relationship or any of its attributes.\(^2\)

\(^2\)Note that relationships, which represent arbitrary “is-associated-with” connections, are semantically different from edges, which specifically represent “is-reachable-from” relationships. Thus, for
Relations are unordered collections of relationships\(^3\) or edges. In Pleiades, relations are defined over a single type of relationship or edge. Relations have multiset semantics—that is, the same relationship or edge may occur multiple times in a given relation. Because relationships have identity, the same relationship or edge may also occur in multiple relations. The relation abstraction also supports the definition of subrelations. A subrelation is a relation whose elements are constrained to be a subset of those in another relation. We have found that a number of different kinds of software engineering applications require the ability to represent and enforce subset and superset semantics; in the example presented in Section 2.1, for instance, some kinds of program dependence information are actually subsets of other kinds \([83]\). Thus, Pleiades supports the definition of subrelations to support such applications.

Instances of relation types are first-class objects with identity. This feature permits the explicit representation of, for example, relationships between relations, and it permits the construction of other types with relations as components.

Operations defined on relation types include ones to create, destroy, insert into, remove from, retrieve from, iterate over, and query relations. Facilities are provided to obtain the difference between two relations (i.e., the set of relationships or edges that occur in one relation but not the other) and to compute the union and intersection of two relations. Metadata about the type of a given relation and its attributes is available.

The relation abstraction supports the definition of indices for efficient querying and for ordered iteration over relations. Indices can be any type, and indexed values need not be unique. For indices whose types are strings or numeric types, indices can be built without programmer intervention. For other types of indices, programmers

---

\(^3\)Relational database terminology defines relations as tables \([30]\). We use the more generic term \textit{collection} to describe Pleiades relations because of the semantic differences between the two relation models.
may optionally specify certain kinds of information about the type to permit Pleiades to construct efficient indices. Currently, Pleiades permits programmers to specify a hash function and/or ordering functions for a given type. It also allows programmers to indicate whether index values are expected to be sparse or dense to guide Pleiades’ selection of an index data structure.

Figure 4.1 shows some Pleiades type definitions for the data structures depicted in Figure 2.2. The nodes in the abstract syntax graph are represented by type AST_Node. The control flow graph includes different kinds of nodes to represent different kinds of control structures; for example, type CFG_If_Node represents conditional branches (e.g., “if” statements). The edges of CFG nodes may have to be annotated, so they are represented as explicit, attributed edges (e.g., type If_Node_Edge). Finally, the connection between nodes in the CFG and the AST nodes from which they are created is represented with the relationship type AST_To_CFG_Relationship. Instances of this relationship may be collected in a relation (type AST_To_CFG_Relation).

We note that the set of type constructors that Pleiades provides was motivated by the requirements of various clients. We recognize that this set is not complete, however. In particular, several other kinds of collection types, such as (ordered and unordered) sets, lists, bags, and directories, have also been found to be useful (e.g., [8]), and support for these kinds of type constructors is planned for a future version of Pleiades.

**Related Work:** Most conventional relational databases, programming languages, database programming languages, and object-oriented databases are limited in their support for graph, sequence, relation, and relationship types. Existing systems do not provide high-level constructors for defining graph types and require programmers to define graph abstractions using lower-level constructs, which is, as noted earlier, a time-consuming and error-prone process that may result in code that is difficult to


type CFG_Statement_Node, CFG_If_Node;
type If_Node_Edge is
  edge from CFG_If_Node
  to CFG_Statement_Node
  Edge Information : Information_Type;
end edge;

type CFG_If_Node is node
  Then_Branch : If_Node_Edge;
  Else_Branch : If_Node_Edge;
end node;

type Statement_Kinds is ( IfStmt,
  ForStmt, ... );

type CFG_Statement_Node is node
  Kind_of_Statement : Statement_Kinds;
  ...
end node;

type AST_Node;
type AST_Node_Sequence is
  sequence of AST_Node;
type AST_Node is node
  Label : String;
  Children : AST_Node_Sequence;
end node;

type AST_To_CFG_Relationship is
  relationship
  AST_Source_Node : AST_Node;
  Associated_CFG_Node : CFG_Statement_Node;
end relationship;

type AST_To_CFG_Relation is
  relation of AST_To_CFG_Relationship;

Figure 4.1 Partial PLEIADES Type Definitions for Example in Figure 2.2.
understand and maintain. The Gras [78] and IDL [48] systems attempted to address this limitation by providing support for the definition of graph types, but neither supports the definition of attributed edges.

Support for varying-length ordered sequences of entities is also surprisingly limited in existing systems. Relational database systems support varying-sized collections in the form of relations, but these collections are not ordered. Applications can achieve the effect of ordered sequences by defining a unique (key) index field, but the maintenance of this field is left to the application. Further, since relations apply only to relationships, the definition of, for example, varying-length integer arrays is not supported directly and must be simulated with appropriate relation definitions. Programming languages generally support ordered sequences with array constructors or with linked lists, but most popular languages (e.g., Ada [121], C [52], C++ [103]) do not support varying-length arrays directly—application code must simulate varying-length arrays using other types. Object-oriented databases often provide array constructors (e.g., ONTOS [5] and the Texas Instruments Open Object-Oriented Database (TIOOODB) [124]), but these are generally limited in the same ways that programming language arrays are (two notable exceptions are Gemstone [66], which provides an indexed set class, and EXTRA [122], which provides an explicit varying-length array constructor that does not support insertion but that does support tail expansion).

Finally, support for relations and relationships is highly variable among existing systems. Relational databases do, of course, support both types of objects. The relational database model of support is not appropriate for software engineering environments [15]. In particular, the fact that relations must be normalized (i.e., they may not have fields whose types are compound objects) is a significant problem in the context of software development environments, where connections between highly complex, structured objects, such as CFG and AST nodes, must exist. Since the kinds of connections that can be represented in the relational model are limited, the
implementations of structured object types whose instances may participate in relationships are correspondingly limited—such types must be implemented as relations. This may lead to implementations that are less efficient, more difficult to understand and maintain, and that incur higher impact of change. We have also found that the lack of identity in the semantics of relations and relationships that database systems define is inappropriate in many software engineering applications, where both types of objects may have to be shared by numerous other objects.

Programming languages do not typically support either relation or relationship constructors directly. This shortcoming, in part, led to the development of database programming languages [7], which provide some form of relation and relationship constructor, but these types are not always fully integrated into the languages—for example, some are not type-complete (e.g., APPL/A [105], Pascal/R [92], ADAPLEX [96], and the Object Database Standard [8]). In addition, the models of relations and relationships that are provided usually have the same restrictions as the relational database model. Object-oriented databases usually only support either relations or some sort of set constructor (e.g., Gemstone [66], O2 [33], ORION [54, 55], EXTRA [122], and TIOOODB [124]). Finally, we note that recent work on data structure precompilers to support software reuse [94] has suggested that the lack of a collection constructor in programming languages reduces the ability to produce highly reusable software components. This work attempted to address this shortcoming, in a system called Predator, by extending the C language with the higher-level collection type constructors list, array, and binary tree. The interfaces to the resulting abstract data types are very similar to those of relational databases. Predator’s collections are limited to collections of C structs, however, and so they have many of the same limitations as database relations.
**Pleiades Limitations and Future Directions:** At present, the Pleiades type model does not satisfy all of the cross-cutting requirements. Because of Ada limitations, Pleiades does not provide procedures and functions as first-class types; these entities cannot, for example, be passed as arguments to operations and they cannot be used as an operand of a type constructor. The Pleiades type model also is not currently fully type complete. In particular, it has some restrictions on the definition of relations—as noted earlier, the relation constructor can be applied only to a relationship or edge type. This is too limiting, since a need for collections of other kinds of objects is pervasive. In addition, relations cannot be heterogeneous—they may contain instances of only a single type of relationship or edge. We imposed this restriction initially because it is difficult to define queries or support indices over heterogeneous relations, but as our research progresses, we hope to remove it. Finally, the query facilities provided are currently limited—a general-purpose query language has not been incorporated.

### 4.1.2 Navigational and Associative Access

As discussed in Section 2.2, software engineering applications may want to traverse data structures navigationally or associatively. Pleiades therefore supports both navigational and associative access.

**Navigational Access:** In addition to the inherent support for navigational access that Ada (and most programming languages) provides, Pleiades provides navigational access to nodes, edges, and relationships through the definition of operations to retrieve attribute values, and it supports navigation over relations and sequences by providing iteration operations. Figure 4.2a shows a Pleiades code fragment that declares instances of the `CFG.IfNode` and `If.Node.Edge` types, shown in Figure 4.1, and then performs a simple navigational traversal of the resulting CFG if-node.
If_Test : CFG.If_Node;
Else_Branch_Node : If_Node.Edge;
...
Else_Branch_Node := Get.Edge ( If_Test, Else_Branch );

(a) Navigational Access.

AST.If_Node : AST_Node;
-- The root of the AST representation for the if-test
AST_To_CFG_Connections : AST_To_CFG_Relation;
-- A collection of relationships between corresponding
-- AST and CFG nodes
If_Node.Relationship : AST_To_CFG_Relationship;
-- The relationship between the AST and CFG
-- representations of the if-test
...
-- Associative Access:
If_Node.Relationship :=
   Select_Tuple ( AST_To_CFG_Connections, Associated_CFG_Node, If_Test );
-- Navigational Access:
AST.If_Node := Get_Attribute ( If_Node.Relationship, AST_Source_Node );

(b) Associative and Navigational Access.

Figure 4.2 Pleiades Code for Navigational and Associative Access to Instances of Types Defined in Figure 4.1.
**Associative Access:** PLEIADES supports associative access over relations through a set of query operations. Both relationships and edges can be placed in relations, so a combination of associative and navigational access can be achieved over these types of objects. Figure 4.2b shows a PLEIADES code fragment in which the AST node from which a given CFG node was created is located. The relation is accessed associatively to find the relationship whose `Associated_Edge_Node` attribute value is `If_Test`. This relationship is then accessed navigationally to retrieve the associated AST node.

**Related Work:** Programming languages support navigational access as a matter of course (e.g., by following pointers or fields of records), but imperative programming languages do not directly support associative access, and rule-based languages, such as Prolog [123], include only a limited notion of associative access. Relational databases, on the other hand, provide associative access (over relations), but they do not directly support navigational access—developers must implement this capability using queries.

Both database programming languages and object-oriented databases have tended to include a dichotomy between types that can be accessed associatively and types that can be accessed navigationally—it would be difficult, for example, to define CFGs to achieve both associative and navigational access in systems such as AP5 [29] and O2 [11]. Some systems, such as Gemstone [66] and EXTRA [122], support set constructors over any type of object, which can be used to allow a programmer to implement these semantics, but they do not directly support navigational and associative access over the same structures.

**PLEIADES Limitations and Future Directions:** While PLEIADES provides navigational access over most of the types it supports, it only provides special support for accessing relations associatively. We intend to extend the kinds of types over which associative access is supported. For instance, the need to query graphs (e.g.,
to find out where all variable definitions occur) is common. In addition, Pleiades does not support the ability to switch between associative and navigational access patterns dynamically, which is a capability that many software engineering applications require. Applications must therefore anticipate that they will require both access methods and maintain separate data structures for use during associative or navigational accesses. Clearly, this is not desirable. We plan to explore strategies for automatically supporting both navigation and associative access over the same structures—e.g., internally transforming between multiple representations of an object to optimize a particular access pattern, optimizing a single representation, etc. It is likely that different approaches will be more useful for different applications, and we hope to provide a framework in which developers can select the strategy that best accommodates their needs. We will also explore the tradeoff between query optimizability and generality to determine the extent to which associative accesses can be optimized in the presence of type and computational completeness.

4.1.3 Persistence

Pleiades defines persistence to be a property of instances, and this property is orthogonal to other properties of the instance [7]. Orthogonality means that the interfaces to persistent objects are identical to those of non-persistent objects [126], so that, for example, queries and concurrency control can occur over both persistent and transient objects. Applications can dynamically select objects that should become persistent. Pleiades defines operations on each abstract data type to make instances persistent and to retrieve persistent objects.

The Pleiades model of persistence is reachability-based—any object that is reachable from, or contained within, an object that becomes persistent itself becomes persistent [126]. Similarly, when a persistent object is retrieved from persistent storage, all of the objects reachable from it become available with no additional application
-- Make the root of an AST persistent:
AST_Root, AST_Retrieved_Root : AST_Node;
AST_Root_Persistent_Identifier : PID;
...
Get_PID ( The_Object => AST_Root,
         The_PID => AST_Root_Persistent_Identifier );

-- Retrieve the root of the persistent AST later:
Get_NPR ( The_PID => AST_Root_Persistent_Identifier,
         The_Object_Reference => AST_Retrieved_Root );
Print ( "Label of root is: " & Get_Attribute ( AST_Retrieved_Root, Label ) );

-- The transitive closure of AST_Root is now (logically) available for traversal.

Figure 4.3 Using the PLEIADES Persistence Interface.

intervention.

Figure 4.3 demonstrates the use of the PLEIADES persistence mechanism. The root of the abstract syntax graph, AST_Root, becomes (logically) persistent after the call to Get_PID, as do all nodes reachable from it. When the root is retrieved from the persistent store (using operation Get_NPR), the graph can be traversed using the usual graph manipulation operations.

The reachability-based model of persistence has proven to be especially suitable for use in software engineering environments, where many of the objects created are connected structures. It is not always appropriate, however. In particular, we have found that a reachability definition based on the “is reachable from” relationship sometimes causes more objects to become persistent than desirable. Indeed, it is not difficult to see how the interconnectedness of objects in software engineering environments could lead to a situation where making any object persistent results in a very large number of other objects becoming persistent. Therefore, PLEIADES currently provides a mechanism at the type-definition level that allows the abstract

\[^{4}\text{PLEIADES logically retrieves the transitive closure of an object, but to minimize the cost of retrieving objects that are not used, it does not physically retrieve any object until an application attempts to read that object.}\]
data type developer to indicate which of the subcomponents of a given type might not become persistent. Attributes that represent potential “cut points” for reachability-based persistence are specified using relationships (where the attribute and the object to which it is related are the fields of this relationship), which changes the “is part of” relationship to “is associated with” [116]. Attributes specified as “associated with” a persistent object do not become persistent by reachability. Thus, applications gain dynamic control over the persistence of these attributes, though only to the extent that an application can determine statically where the potential “cut points” will be. For example, although the code shown in Figure 4.3 will cause the entire abstract syntax graph of Figure 2.2 to become persistent, neither the relationships between AST nodes and CFG nodes nor any CFG nodes will become persistent—persistence has been limited by the use of relationships between AST and CFG nodes instead of (explicit or implicit) edges. Of course, an application that uses this abstract data type has the ability to make the “associated” subcomponents of any instance persistent as well; for example, making the relation \( \text{AST-To-CFG-Connections} \) (shown in Figure 4.2) persistent would cause all of the relationships between AST and CFG nodes to become persistent by reachability, and in turn, all of the AST and CFG nodes that are related (and their transitive closures) would also become persistent.

The Pleiades model of persistence satisfies the requirement for generality in that it readily supports other commonly used models of persistence. For example, models in which persistence is determined by type (i.e., all instances of a given type become persistent; e.g., E [88]) and where all instances become persistent (e.g., Gemstone [21]) are readily modeled using a persistence-by-instance model—the “make persistent” operation is simply called from the appropriate “create object” operations. To achieve persistence-by-instance using either of these models is considerably more difficult, however, since it requires an application to keep track of and destroy all the objects it does not want to persist. Similarly, the Pleiades model satisfies the
requirement for dynamic control over persistence. Finally, while reachability-based extent of persistence has proven to be appropriate for many software engineering applications, Pleiades supports the definition of alternate paradigms.

**Related Work:** Traditional programming languages are generally limited in the ways in which they support persistence—they normally provide only a file abstraction, which requires a considerable amount of programmer effort to “flatten” structured data and save them in a file. Further, type integrity cannot be enforced once objects have been saved to a file. Relational databases have just the opposite problem—they make *all* relations (and only relations) persistent automatically and provide applications with no control over what becomes persistent, thus violating the cross-cutting requirements for dynamism and completeness.

Database programming languages (e.g., [92, 105, 77, 88]) often provide persistence only over database types, and therefore violate the completeness requirement in much the same way that relational databases do. Some exceptions are [6, 32, 40], which provide dynamic control over the persistence decision, but not over extent of persistence, and the Ergo system [58], which provides static (but not dynamic) programmer control over extent of persistence (i.e., the definition of a type must designate components as persistent or non-persistent). Object-oriented databases, on the other hand, support persistence over a wider range of types. Most object-oriented database systems either assume that all objects persist and do not provide applications with control over which objects become persistent (e.g., [4, 122, 66]), tie persistence to types (e.g., O2[11]), or limit the types of objects that can be designated as “top-level” persistent objects (e.g., [62]); one notable exception is TIOOODB [124], which supports dynamic persistence decision over any type of object, but again, this system does not support control over extent of persistence, and it does not correctly handle identity of objects (i.e., if two objects, \( o_1 \) and \( o_2 \), each refer to a third object, \( o_3 \), then
when \( o_1 \) and \( o_2 \) become persistent, each will receive its own copy of \( o_3 \), rather than continuing to share it).

**PLEIADES Limitations and Future Directions:** PLEIADES does not yet satisfy the cross-cutting requirement for completeness—the current implementation only automates the generation of the persistence mechanism for graphs, sequences, relations, and relationships. We have anticipated the inclusion of persistence for all types of objects, however, by implementing persistence through a general persistence protocol. Supporting persistence for other types is achieved simply by providing \texttt{Get_PID} and \texttt{Get_NPR} operations on the types, and we have done this manually for a number of types. PLEIADES does not yet satisfy the requirement for dynamic control over extent of persistence to the degree that we believe would be desirable. The current mechanism for limiting the extent of persistence requires the abstract data type implementor to determine potential "cut points" statically and to base the selection of type constructor on them, which does not always result in the cleanest or most natural representation for a given abstract data type. We plan to extend PLEIADES to permit applications to indicate dynamically that a subcomponent of a given persistent object should not become persistent so that decisions about type definitions need not be affected by persistence concerns. Finally, support for deletion semantics is currently limited to an unchecked (and thus, unsafe) "destroy" operation. We plan to explore desirable semantics for identifying objects that are no longer useful or meaningful (for example, a symbol table is not likely to be useful once the module with which it was associated is destroyed) and appropriate implementation strategies.

### 4.1.4 Consistency Management

As described earlier, object management support for software engineering must be able to detect and react to a range of different kinds of violations of different consistency definitions. For example, some violations must be precluded (e.g., type
violations), while others may be allowed to occur and rolled back if they are not eventually corrected, and still others may be allowed to occur and “rolled forward” to a new state that satisfies the violated consistency definition.

Consistency definition is supported in PLEIADES by the specification of constraints. A constraint is a Boolean operation that tests for the satisfaction of some condition. Constraints are computationally complete, so any condition for consistency can be specified. Constraints are dynamically and statically enforceable and relaxable, and they can be enforced or relaxed on a per-instance or per-type basis. PLEIADES provides operations to enforce and relax constraints over graph, sequence, relation, and relationship types. To enforce a constraint, an application must specify a set of operations in which the constraint might be violated, and an optional action that is to be taken upon detection of the violation (by default, an exception is raised). Actions, like constraints, are computationally complete, so any required action may occur in response to a constraint violation. Constraints are checked in any operation in which the application indicates that they may be violated, and they can be checked as a precondition, postcondition, or both, depending on the semantics the application requires. An application can test for the satisfaction of any constraint at any time, whether or not the constraint is enforced. The PLEIADES model of dynamic enforcement of constraints is based on APPL/A’s [109].

The PLEIADES consistency management model satisfies the cross-cutting requirement for generality—it supports the detection of consistency violations either before or after they have occurred (as desired), and it supports the definition of a variety of consistency reestablishment mechanisms, including violation preclusion, roll-forward, and roll-backward, so it facilitates multiple consistency management mechanisms. It

---

5In relational database terminology, this is called a predicate.
6Note that because PLEIADES employs an abstract data type model, it is only possible to modify the state of an object, and thus to violate an enforced constraint, by invoking an operation on the object or one of its subcomponents.
7PLEIADES should not be confused with constraint-based programming languages (e.g., [19, 98]), which support problem-solving through the definition and satisfaction of sets of constraints.
also satisfies the requirements for completeness and dynamism to some extent—both constraint and action definition are computationally complete, and control over enforcement of constraints and actions to be taken upon constraint violation can occur dynamically. Finally, meta-data is provided in the form of operations that determine dynamically which constraints are currently enforced on a given object.

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-date requirements, etc.). Responses to potential violations are usually limited to raising an exception exception (e.g., [121, 61]). In addition, programming languages typically support only preclusion semantics—they will prevent consistency violations from occurring, but they do not support roll-forward or roll-backward semantics without programmer intervention. Assertion (e.g., [90, 65]) and exception handling mechanisms (as in Ada [121] and CLU [61]) 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 of dynamic control over attachment of an assertion or enforcement or first class status or identity of the conditions or actions associated with these mechanisms.

8PL/1 ON conditions do support dynamic enforcement.
Many relational database systems support the definition of constraints. Their constraint enforcement mechanism does 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 either do not support consistency management (e.g., [122, 4]), support limited consistency definitions, such as referential integrity (e.g., [66]) or programming-language-style consistency definitions (e.g., [4]), or support consistency definitions over only a subset of types (typically collection types; e.g., [105, 29]). A few active database systems, such as [99, 22, 62, 12], have included better support for constraint definition, but these systems have had a variety of limitations (e.g., [22] does not support programmer-specified actions).

Consistency definition mechanisms in relational database, database programming language, and object-oriented database systems fail to satisfy the cross-cutting requirement for first-class status of constraints, which means, for example, that information cannot be associated with constraints, and that information about the relationships between constraints cannot be encoded explicitly, making these relationships difficult to comprehend and maintain. A notable exception is HiPAC [31], which defines rules to be first-class objects.

**Pleiades Limitations and Future Directions:** The current implementation of Pleiades does not satisfy the cross-cutting requirement for first-class status for constraints because Ada does not treat operations as first-class entities. Pleiades also does not yet entirely satisfy the requirement for dynamism. While control
over constraint enforcement can occur dynamically, constraint definition must occur statically—new constraints cannot be defined dynamically. Similarly, while association of actions with constraints can be done dynamically, the definition of all possible actions must be defined statically. These restrictions are artifacts of Ada’s static type model, so they cannot be overcome readily. Finally, the requirement for completeness has not yet been satisfied, since consistency definitions can be enforced only over graph, sequence, relation, and relationship types. This restriction is expected to be removed in the future.

We have found that consistency management is complicated by the cross-cutting requirement for object identity. The state of a shared object may affect that of any object that refers to it; for example, the state of a set depends on the states of each of the elements contained within it, and as those states change, the state of the set changes. When an object is sharable, it can be manipulated independently of any objects that refer to it. This may lead to situations in which one object that refers to another can enter an inconsistent state indirectly (i.e., even though none of the operations defined on it have been invoked). We have been exploring mechanisms for addressing this consistency management problem.

4.2 Conclusions and Future Work

PLEIADES provides a number of useful object management capabilities. Our approach to developing this object management system has been to consider the demanding needs of software engineering environments and to attempt to formulate both the functional and cross-cutting requirements imposed by this domain. The result has been a system that provides many of the capabilities found in a database system but provided in a style that has more of a programming language flavor. PLEIADES provides an abstract data type model of object management, where object

---

9This issue is discussed fully in Chapter 7.2.
management capabilities such as consistency constraints and persistence are provided as “inherited” operations on any type.

Although Pleiades provides considerable benefit, there are many ways in which we intend to improve the system. As noted in the previous section, there are some capabilities that do not support our cross-cutting requirements as fully as we would like. We are currently exploring ways to address these limitations. Also, as noted in Section 2, there are some areas of functionality that we have yet to address. We have done preliminary investigation into most of these areas (e.g., [113]), and, in some cases, we have built independent prototypes [130, 59]. Although there is considerable interaction among the current and proposed capabilities, we feel relatively confident that our current system design will readily support these extensions.

Pleiades has been used in about half a dozen projects within the Arcadia consortium (e.g., [87]), as well as by some industrial users (e.g., [64]). For the most part, users of the system have been very pleased with the functionality of the system and its overall performance. A complete evaluation of the use of Pleiades in some of these systems is presented in Chapter 8.
CHAPTER 5
INSTANTIATING THE PERSISTENCE MODEL

Instantiating the framework described in Chapter 3 for a programming language in general, and for Ada (in the form of Pleiades) in particular, involved addressing a number of design and implementation decisions and tradeoffs. The most interesting issues we discovered were in producing instantiations of the persistence, consistency management, and concurrency control models. Thus, this chapter describes issues pertaining to instantiating the persistence model, while the next two chapters discuss those pertaining to consistency management and concurrency control, respectively.

5.1 Design-Level Issues in Mapping the Framework to Ada

The framework described in Chapter 3.2 specifies that persistence is modeled via a persistence attribute, which is a first-class object with identity. The persistence attribute is manipulated through three accessor operations: Make_Persistent, which sets the attribute’s value to True, Make_Transient, which sets the attribute’s value to False, and Is_Persistent, which indicates whether or not a given object is persistent. Further, extent of persistence is defined as the set of objects that share the same persistence attribute.

5.1.1 Satisfying the Cross-Cutting Requirements

From a design perspective, the framework maps quite readily to Ada. It is not difficult to define a persistence attribute on objects, and this was, in fact, how persistence was designed in Pleiades. Satisfying the cross-cutting requirement for
identity of the persistence attribute, however, turned out to raise some design- and implementation-level concerns with respect to the definition of the required semantics of the `Make_Persistent`, `Make_Transient`, and `Is_Persistent` operations.

Implementing the persistence attribute as a first-class attribute with identity makes defining the `Is_Persistent` operation trivial—`Is_Persistent` need only return the current value of any object’s persistence attribute. The implementations of `Make_Persistent` and `Make_Transient` are more difficult, however. While actually setting the persistence attribute’s value is trivial, the process of physically making a transient object persistent requires allocating space on disk, moving the object to the persistent heap, etc.; similarly, making a persistent object transient requires deallocating persistent storage and cleaning up persistent data structures. When a persistence attribute has identity, it may be shared by multiple objects—in fact, the ability to control the extent of persistence in this manner is one of the most significant benefits of satisfying the identity cross-cutting requirement. This means, however, that when the value of a persistence attribute changes, the change affects the persistence status of the entire set of objects that share the attribute. Thus, when the value of a persistence attribute changes, it is necessary to determine which objects share that attribute so that the actions associated with becoming persistent or transient can be applied to this set of objects. This implies that it must be possible to navigate from each persistence attribute object to all of the objects that share it, which is not explicitly part of the semantics of persistence attributes described in Chapter 3.2.

Clearly, numerous implementation options are available to address this problem, including back-pointers from each persistence attribute to all of the objects that share it, notification, and periodic scans of the object space (similar to techniques used in garbage collection).1 Back-pointers are easy to implement, but their use comes at a fairly high cost, both in terms of space (the extent of an object could be quite large,  

1This problem is actually a form of the container problem, which is described in Chapter 7.
as in the case of a graph, which would make the set of back-pointers correspondingly large) and time (i.e., upon each object update, the set of back-pointers associated with the object’s persistence attribute would have to be updated). Event-based notification techniques do not scale well. In the presence of a garbage collector, it might be feasible to piggyback the detection of changes in persistence status with the garbage collection process, but we did not assume the presence of a garbage collector. Without one, a scanning technique requires identification of a set of “root” objects from which to start scanning. Since any object can become persistent at any time during its lifetime, it is difficult to identify such a set, which would make scanning quite costly.

At the time we designed Pleiades, we were concerned about the performance costs associated with a design in which persistence attributes have identity, since we were unaware of any scalable, distributable algorithms for identifying the set of objects affected by a change to a persistence attribute’s value. These concerns essentially precluded the direct mapping of the framework to an Ada implementation. Thus, we opted for an alternative design, in which the persistence attribute is modeled as a first-class entity, but without identity, which means that each object has its own persistence attribute. A straightforward algorithm exists for computing the extent of persistence: **Make_Persistent** designates the set of persistent “roots,” and all reachable objects can be found by traversing from that set of roots.

Using this design obviously does not satisfy the cross-cutting requirement for identity, which has some ramifications on support for control (both dynamic and static) over extent of persistence. Since the framework models extent as the set of objects that share the same persistence attribute, omitting identity meant that we had to add to the design of Pleiades a different mechanism for supporting explicit control over extent of persistence. Since complex objects are so often highly connected,

\[\text{Since that time, advances in other areas, particularly memory management (e.g., [47]), have produced techniques and algorithms that may provide the basis for providing adequate performance for persistence attributes with identity. We hope to explore the application of these techniques as future work.}\]
interrelated structures, we believed we could provide adequate support by providing support for commonly used definitions of extent of persistence, along with override mechanisms. The most common definition of “extent” is by navigational reachability, as found, for example, in graphs, lists, and collections, so we used it as our default. Override mechanisms are provided that permit control over extent of persistence specified at either the type or instance level, and these mechanisms are described below. We note that the failure to satisfy the cross-cutting requirement for identity has a significant impact on the mapping from the framework into a programming language.

5.1.1.1 Type-Level Control Over Extent of Persistence

Two forms of control at the type level are provided. One is the use of transient subcomponents. References to any subcomponents of an object that are designated transient do not become persistent as a result of navigational reachability. This mechanism provides only static control over extent of persistence, unlike the use of relationships, but it is useful in cases where it is known a priori that parts of an object are inherently transient. This is similar to the static control mechanism provided in the Ergo system [58], in which components are explicitly designated as “persistent” or “non-persistent,” except that the static control is only over non-persistent components—dynamic control still applies to the selection of persistent (and non-persistent) objects. This solution does violate, to some extent, the orthogonality property that we have hypothesized to be necessary, in that transient objects do not have all the same operations as any other objects (in particular, Make_Persistent and Make_Transient are not defined on such objects).

The other type-level control mechanism for overriding navigational reachability is based on the observation [116] that objects often have two kinds of parts: core parts and associated parts. Core parts are those components of an object that must always be present for the object to have any meaning. For example, any node in
a binary tree must, by definition, have data and must have (possibly null) left and right children. Associated parts are pieces of information that some instances of a type have while others do not. For example, a node in a graph that is being depicted on a display device might have associated with it the coordinates on the display at which the node appears, but this information does not exist for nodes that are not displayed. Associated information is not less important than core data—the difference, for example, is that the user interface utility does not expect every node to have position information, and nodes that do not have it are still meaningful.

This distinction provides a basis for principled introduction of “bounds” or “limits” in our persistence mechanism. We take the position that core attributes must persist if the object to which they belong persists, and so should be subject to a reachability-based persistence model. Associated parts, on the other hand, need not persist. The user interface utility, for example, might or might not need to preserve any or all of the position information it creates, depending perhaps on whether or not the user asked to have some parts (or all) of the display saved for use at a later time. Therefore, associated parts should be potentially persistent—that is, they can be made persistent explicitly, but they will not become persistent automatically simply because the object with which they are associated becomes persistent.

Thus, the other type-level control mechanism for overriding navigational reachability is to use relationships to represent connections among objects that may or may not become persistent, rather than embedded references (using edges or object identifiers). Relationships, as described in Chapter 4, are not navigationally reachable, unless they are directly embedded in an object. Thus, representing connections among objects with relationships provides a mechanism to define the boundaries that delimit persistence in complex, connected structures. The connections can then be made persistent selectively through the use of the standard \texttt{MakePersistent} operation. We have described this mechanism in detail in [116].
5.1.1.2 Instance-Level Control Over Extent of Persistence

For instance-level control, we specified two mechanisms for Pleiades\(^3\). The first is a special form of the Create operation, which allows objects to be designated as persistent or transient upon creation:

\[
\text{function Create ( IsPersistent : Boolean )} \\
\text{return Object;}
\]

The other is an operation that allows objects to be designated as “cut points” for persistence by navigational reachability—that is, objects that do not become persistent, even if they are navigationally reachable from persistent objects:

\[
\text{procedure Designate_Cutpoint ( O : in out Object );}
\]

5.1.2 Naming Issues

Naming issues are somewhat more complicated in the presence of persistence than they are in transient programs. While all programming languages provide mechanisms for referring to instances of transient types (“names”), the validity of these references is not usually guaranteed outside of a single program execution. The ability to define the `RetrievePersistentObject` operation, however, depends on the ability to have designators for persistent objects that are valid across program executions. Thus, the definition of a persistent name space is a necessary part of any instantiation of the persistence part of the framework. In order to achieve a name space of object references that is valid within and between separate program executions, one must gain control over the name space in one of two ways: by modifying the run-time system of the language or by custom-building a name space on top of that which is already provided by the language. In fact, the first approach is the one taken in most persistent programming languages (e.g., [23, 42]), while the second is often found in object-oriented databases (e.g., [8]).

\(^3\)These mechanisms are not implemented in the current version of Pleiades, however.
Our desire to satisfy the heterogeneity cross-cutting requirement suggested to us that it was dangerous to assume a “single” name space that is under the control of a single object management system. The heterogeneity requirement is made because we assume that objects defined using one object management system may have to be referred to by objects defined using other object management systems (since, as discussed earlier, different applications will be best served by making different tradeoffs in instantiating the framework, which leads us to believe that no single object management system will satisfy the needs of all applications). Thus, we believed it was important to define a “persistent” name space in such a way that references defined in this space can be used safely to refer to persistent objects across executions and across object management systems.

As a result, we chose to use two, side-by-side name spaces, and to translate between them. One name space contains the “normal” object references, or object identifiers (OIDs), that are defined by the host language. These references cannot be preserved beyond an application’s execution. The other name space contains references to persistent objects. We call references in this name space persistent identifiers (PIDs). When applications manipulate objects (persistent or transient), they use OIDs. PIDs are used solely as a means for retaining references to persistent objects, and for retrieving persistent objects. When a persistent object is retrieved, the application is provided with the OID for that object, which it uses subsequently. Thus, we added the following operations to those defined to support persistence:

```plaintext
function Get_PID ( O : in Object ) return PID;
function Get_OID ( P : in PID ) return Object;
```

Note that Get_OID subsumes the functionality of Retrieve_Persistent_Object; in fact, their signatures and semantics are identical, except that type PID is used in place of the untyped String in Retrieve_Persistent_Object. Further, note that Get_PID subsumes Make_Persistent, since an object must persistent for as long as a persistent reference to it exists.
We note here that this mapping to Ada does not address a number of important issues in name space management, such as scoping, hierarchy, etc. These issues, while important, were not the primary focus of this work, and they have been addressed in much more detail in other work (e.g., [51]). Our major concern in producing an instantiation of the framework was solely to provide enough of a name space solution to allow us to evaluate our hypothesis.

5.2 Implementation Issues

Producing an implementation of the framework required making certain tradeoffs. We summarize some of these below.

5.2.1 Implementation Architecture

One of our stated goals in this work is to facilitate many forms of heterogeneity. One obviously useful form of heterogeneity is to permit the use of alternative underlying support systems, such as storage managers, concurrency controllers, etc., since any given support system is likely to provide different performance characteristics that make them more or less suitable to a given application. Thus, in designing an implementation architecture, we wanted to make it straightforward for application developers to use alternative support systems. Clearly, if changing underlying support systems necessitates recoding of existing applications, developers will be reluctant to incorporate new technology. Even paying the lesser price of large-scale recompilations is often unacceptable to developers. We had the goal, therefore, of defining an implementation architecture in which we could limit the cost of swapping underlying support systems to a simple relink; at worst, we were willing to tolerate a compilation cost as long as it did not require the recoding of any existing clients.

To achieve this goal, the implementation of Pleiades interface packages is based on a layered architecture that minimizes the impact of changing the underlying sup-
Figure 5.1 Anatomy of a PLEIADES interface package.
port systems, as shown in Figure 5.1. Each interface package is built on a Storage Manager Interface (SMI) [114], which defines a standard interface to required low-level persistent storage capabilities. The SMI can be implemented on any commercial or research persistent storage system, and different Pleiades clients can freely and transparently use any (or multiple) low-level systems they require. The SMI specification itself is constant, but the specification can provide multiple implementations. In this way, the peculiarities of particular instances of low-level systems are hidden from application programs by interface packages, and from interface packages by the SMI.

In many cases, the architecture we defined allows clients a link-time choice of low-level support system.4 In some cases, however, certain types that must be defined in the SMI specification (i.e., the implementations of Ada private types) must have different representations for different storage systems. For example, a record might have to contain additional fields to keep track of information that is significant to one low-level system, but not to another. Such changes are confined to the private part of the SMI, which means they do not impact any clients code. A change to the private part, however, requires a recompilation of the SMI specification, which requires the recompilation of client code as well.

The original version of the SMI specification was based in large part on the Mneme system [74], which we felt provided a very general-purpose storage manager specification. As a result of later experimentation, we have revised the SMI specification to provide a somewhat higher-level abstraction than the original version.

5.2.2 Minimizing Read/Write Costs

Reading from, and writing to, disks is expensive. The problem of when to move persistent objects is complicated by the fact that applications may manipulate ob-

4In languages, like Java, that provide more dynamic control than Ada does, it would have allowed a run-time choice.
ject spaces larger than the size of virtual memory, meaning that the same object may have to be read and written multiple times per application execution. Thus, a significant implementation concern in constructing any persistent object system is how to minimize the number of disk accesses, and a considerable amount of existing and ongoing research attempts to address some of these issues (e.g., [88, 93, 43]). Some commonly used approaches include compiler optimization, in which various program analysis techniques are used to extract information statically about when, and for how long, particular persistent objects will be needed (e.g., [88]); provision of programmer-supplied directives as “hints” about intended usage patterns (e.g., [93]) or to specify “clusters” of objects that are likely to be accessed together (e.g., [74]); and adaptive algorithms, which record recent access patterns and use caching and fetching strategies that optimize for those patterns in subsequent accesses.

Issues of optimization are significant research problems unto themselves, so they were not within the scope of this thesis—we intend to take advantage of new optimization technology as it becomes available. Instead, our primary concerns are to produce an implementation that minimizes read/write costs as much as possible, given the information we have available to us, with the goal of making performance acceptable (rather than optimal). Thus, the questions that concerned us were when, and how much, to read from a repository when access is requested to a particular object. On one end of the range of possible solutions is an approach in which all objects that are navigationally reachable from an object are retrieved when access to that object is requested by a tool. In this scheme, we would accept the one-time cost of retrieving all objects, but we would then know that all objects were in memory and avoid potentially costly run-time checks for missing objects. On the other end of the spectrum is a completely “demand-driven” scheme, in which objects are retrieved from secondary storage only when they are requested. This ensures the minimum amount of data transfer from secondary storage, as well as minimal consumption of primary
memory, but it incurs the additional cost of run-time checking to determine if a requested object is already in memory. In addition, demand-driven approaches require the ability to detect the first access of an object that has not yet been retrieved from persistent storage, so that the object can be faulted in. Faulting mechanisms may impose additional object access overhead (e.g., if indirection is used, as in Pleiades, it increases the amount of time per object access). The benefits of clustering objects in a persistent store also decrease as the amount of on-demand retrieval increases.

Given our expectation of high contention for objects (discussed in Chapter 2), we selected initially a transparent, demand-driven scheme for retrieval of persistent nodes and collections of nodes, employing surrogates. Surrogates are patterned after the forwarders described in [73]. In essence, a surrogate is a (non-persistent) object that contains the PID of a persistent object and, if the object is resident, an OID for the object. Normal references are then always through surrogates. When Retrieve_Persistent_Object is called the first time for a persistent object, a surrogate is created for that object and the OID for that surrogate is returned. However, the object itself is not retrieved from secondary storage. Instead, the OID in the surrogate is set to a null value, indicating that the object has been requested, but that the information stored in the object has not yet been accessed. The surrogate is placed in a table that is hashed by PID. All objects that reference this persistent object use the surrogate. If an explicit request is made for access to the information contained in the object, then the object is actually retrieved from secondary storage and a reference to it is placed in the surrogate. Note that the use of surrogates is completely hidden from tools. In particular, it appears to a tool using a Pleiades-generated interface package as though it has direct access to persistent objects. Everything navigationally reachable from a retrieved persistent object becomes logically resident, so using the normal component accessor operations (e.g., Get_Attribute) work as expected, but objects may be transparently faulted into memory as they are accessed. We note that
this solution has the additional run-time cost of an extra level of indirection, but the
degree of transparency of access it has enabled has been extremely valuable.

It has been suggested that completely demand-driven methods of object retrieval
typically do not perform well enough (e.g., [15]), so we had always intended to ex-
pand the initial implementation to include a scheme in which a block of objects is
retrieved when any one element of that block is requested. Further, we intended to
allow clients to define the criteria for block membership both statically and dynami-
cally (noting that navigational reachability is not a sufficient criterion, since it does
not, for example, work with disconnected graphs). We performed an experiment in
which we attempted to use information about object types to help guide the selection
of blocking criteria. For example, upon accessing relationships, we retrieved all of the
endpoints of the relationships actively, and upon accessing sequences, we retrieved
the first $n$ elements of the sequence. When this version of Pleiades was used in an
application [35] that had used the fully demand-driven approach, however, the ap-
lication developer reported that the application's performance had degraded by at
least an order of magnitude. Upon closer examination of this application and others,
we discovered some interesting characteristics about the applications. One is that
many applications often traverse only a small number of relationship endpoints. For
example, the application in question was manipulating a relation whose relationships
connected identifiers with the roots of graphs and with other structures. The appli-
cation needed to examine the set of identifiers contained in the relation, but did not
manipulate any of the graphs; thus, the cost of retrieving the roots of these graphs
from persistent storage represented a significant amount of wasted time. Other ap-
lications showed similar access patterns, which suggests that simple strategies like
"retrieve $n$ levels down from this object" are not appropriate. Another characteristic
we noted is that the applications tended to examine either all of the elements of se-
quences, or none of them. Thus, the “retrieve $n$ elements” strategy was a poor one, since the effort is either wasted or insufficient.

As a result of these experiments, we incorporated a fully demand-driven approach to retrieving node and relationship objects; an eager approach to retrieving edges; and a strategy in which we retrieve all members of a sequence upon the first read access of any member. For future work, however, we recognize that application-specific optimizations will be necessary to achieve the best performance.

5.2.3 Centralized vs. Decentralized Control over Persistence

Persistent object systems must designate some component(s) as being responsible for physically reading and writing instances of persistent objects. There are two basic approaches to addressing this issue, as depicted in Figures 5.2 and 5.3. One approach, shown in Figure 5.2, is to designate a (logically) centralized component as being responsible for the reading and writing of persistent objects. This approach is used in most database systems, in which a strong separation between client and server code is made [16]. Clients request reads and writes of objects; servers perform the actual reads and writes of the objects. In this approach, knowledge about the representation of an object may reside in both the client and server.

Given our emphasis on facilitating heterogeneity in the form of multiple persistence models, underlying representations, and implementation strategies, we wanted to use a control mechanism that moves as much control as possible into objects. We refer to this approach, shown in Figure 5.2, as decentralized, because the responsibility for actually moving objects to and from stable storage is left to each type-definition module. In other words, each type is responsible for its own storage. There are two advantages to this approach. First, the implementations of types are hidden; only the type needs to know about how its instances are mapped between in-memory and on-disk representations. Second, using a decentralized approach permits optimizations based on a type’s or instance’s semantics that would be considerably more difficult to
accomplish if a centralized mechanism were employed. For example, using a general-purpose mechanism, a half-filled table would likely occupy as much stable storage and take as much translation and transfer time as a full table. This is because the mechanism cannot be expected to distinguish, in general, meaningful values from meaningless ones (such as those found in the unused part of a table). Under the decentralized approach, the implementor of the type for the table could allow for specialized compaction activities that a general-purpose mechanism could not hope to divine.

The approach taken for PLEIADES’ persistence mechanism is actually a mixture of the two basic approaches. The approach is decentralized in the sense that each PLEIADES-generated interface package is responsible for the storage of its objects. But the approach is centralized in the sense that it embodies a general-purpose mechanism; PLEIADES automates the construction of storage code, freeing developers from this
Figure 5.3 Decentralized approach to storage management.
burden in the same way that a true centralized approach would. In cases where the persistence mechanism selected by Pleiades is not adequate for a given type or object, developers can readily replace the persistence mechanism for the type with one or more alternative mechanisms that are more appropriate.

While the decentralized approach has several advantages with respect to helping to satisfy the heterogeneity cross-cutting requirement, it is clearly not possible to dispense with centralized control entirely. Centralized information is required in a few circumstances: for memory management (i.e., to determine what objects are in memory and when to swap objects in and out of memory), and for transaction management and recovery. With respect to transaction management, one issue that we faced immediately was when to move persistent objects to stable storage—immediately, upon becoming persistent and upon modification to persistent object, or at some later point. The immediate-write model works well in the context of decentralized control, in that it requires no coordination among objects to achieve persistence. Its disadvantage is that it requires writing an object upon each update. Since many applications write to objects multiple times, we were concerned that this strategy would incur enough extra disk accesses that performance would be unacceptable (though in applications that tend to write once, this approach might be a good one). Thus, we decided on a deferred-write approach in our initial Pleiades implementation. In particular, we used the fairly standard database approach of tying the movement of persistent objects to disk until the end of transactions [16].

Deciding that the storage of persistent objects should occur at the end of transactions had another interesting effect on our implementation. Because every type defines the persistence capability for its own instances, storage and transaction management must also be decentralized, in that each type must communicate with a

---

5This issue, as will be discussed later in this chapter, is also related to recovery management, since either choice has implications for the kind of recovery management mechanism (undo or no-undo) that can be defined.
**Figure 5.4** The repository manager abstraction.
low-level storage manager to physically read and write its instances and understand
the notion of “transaction” (if for no other reason than to know when it is expected
to flush its persistent objects to disk). Thus, a “transaction” comprises a set of
transactions running on individual types. Centralized control is therefore needed to
coordinate the individual transaction processing actions of each type. We provided
this control through the definition of a repository manager abstraction, depicted in
Figure 5.4. A repository is essentially a virtual persistent object store—it provides
applications with a view of a single persistent store, even though the repository phsy-
ically comprises one persistent store per ADT. Repositories provide operations to
create, destroy, open, and close repositories, and begin, end, and abort transactions
(the transaction manipulation operations are described in Chapter 6). As in database
systems, persistent objects stored in repositories may be accessed only during a ses-
sion to ensure that multiple accesses to a given object do not conflict. Repository
managers do not themselves perform any direct manipulation of individual persist-
tent stores, however. All manipulation of objects in persistent storage is done by
the ADTs themselves, as discussed earlier. The repository manager simply serves as
a coordinator. When an application wants to manipulate a repository or begin or
end a transaction, it invokes the corresponding operation in the repository manager.
The repository manager, in turn, notifies the ADTs, which communicate with their
storage and transaction management systems as necessary.

5.3 Interaction Issues

Although one of our stated goals for this work was to satisfy a requirement for
orthogonality of functionality, we discovered that both the design and implementation
of a persistence mechanism that satisfies the cross-cutting requirements result in inter-
actions between the persistence semantics and other object management capabilities.
Some of these are described in this section.
5.3.1 Persistence and Type Extent

The *extent* of a type is the set of all instances of the type. Many applications need to know the extent of a type as part of the set of meta-data provided on a type [8], and thus, `Extent_Of` was specified as a required operation on types (see Chapter 3).

In the presence of persistence, and particularly in the presence of multiple repositories, the definition of the extent of a type becomes less clear. When multiple repositories are supported, as they are in Pleiades, instances of a given type of object can be saved in different repositories. In this case, a given application might require different semantics for “extent.” For example, in the case of an abstract syntax graph, different users might have their own repositories, and they might each populate their own repositories with their own abstract syntax graphs. From their perspectives, the extent of the ASG type is the set of ASGs in their own repositories (plus any transient instances they are manipulating). In other cases, applications might need to examine the set of all the ASGs in the world, or all the ASGs in a set of repositories.

Thus, the definition of `Extent_Of` is affected by the presence of persistence.

5.3.2 Dynamic Control of Persistence and First-Class Status of Types

As long as any instance of a type exists, the type must continue to exist. Thus, in the presence of persistent instances of a type, it is not meaningful to allow the type to be destroyed. This suggests that various forms of dynamic control over the extent of persistence should not be permitted in the case of type objects. That is, types are considered reachable from their instances (by the definition of the `Type_Of` operation). Thus, navigational reachability alone could ensure that a type object becomes persistent as soon as one of its instances does. However, the persistence model admits several mechanisms for overriding the navigational reachability default (described earlier in this chapter). Using these mechanisms, it would be possible,
therefore, for an application to make an instance of a type persistent without making
the type persistent. Clearly, these semantics are nonsensical.

In the presence of persistence, it is important for types to be reachable navigationally from their instances (to ensure the correct persistence semantics). This suggests that none of the mechanisms provided for limiting the extent of persistence (e.g., operation \texttt{Designate\_Cutpoint} and the other mechanisms described earlier) should be defined on any object’s type.

5.3.3 Recovery and Persistence

Two general approaches to recovery are used typically [16]: \textit{undo} and \textit{redo}. When recovery is based on undo, changes to persistent objects are written back to the repository before the commit of the transaction in which the changes occurred. In the event of failure or transaction abort, the log is used to restore the original states of all objects that were modified during the transaction. Recovery management systems that are based on redo, in contrast, do not modify objects in the repository until the commit of a transaction that performs the modifications. Thus, if a failure or transaction abort occurs, the modifications are simply discarded. In redo-based recovery, a log of modifications occurs, and, upon transaction commit, the change log is used to propagate the changes to the repository.

As noted earlier in this chapter, implementing the persistence capability required us to decide when objects would actually be written to disk—immediately upon the invocation of the \texttt{Make\_Persistent} operation and upon update to a persistent object, or at some point later during execution. This decision clearly has ramifications for the approach to recovery that can be used, since an immediate-write approach corresponds to undo-based recovery management scheme, while using a deferred approach corresponds to recovery by redo.
5.3.4 Concurrency Control and Persistence

The presence of concurrency control requires the presence of some entity that knows which clients are attempting to use which objects, to preclude conflicting concurrent accesses. This entity, which we will call a concurrency manager, encapsulates state information that is shared across the concurrently executing processes. Thus, it must execute in a separate process space. This means that interacting with the concurrency manager requires interprocess communication (IPC), which comes at a significant cost. The combined cost of both a disk access and an IPC upon each attempt to read or write a persistent object from/to disk produces unacceptable performance. Thus, a significant issue in implementing persistent object management is defining appropriate clusters of persistent objects [16, 12]. Clusters are collections of objects, many of whose members tend to be accessed together. If an application attempts to access any of the members of a cluster, all of the members are brought into memory at the same time, which reduces the number (and consequently, the cost) of disk access and IPCs. Clearly, the larger the size of a cluster, the smaller the cost of storing and retrieving persistent objects.

As has been discussed in the literature, there is an inverse relationship between the amount of concurrent accesses that can occur and the size of a cluster—larger clusters mean less concurrency, and vice-versa. One of the most important aspects of maximizing concurrency and minimizing the cost of persistent object management is producing good clustering schemes [93].

Most current lower-level storage management systems and object-oriented database systems support some means of client-specified clusters. For example, Mneme [74] defines two mechanisms, pools (collections of objects that are are managed by the same policy, which can include control over clustering) and object creation-time hints about related objects, to provide control over clustering. ObjectStore [79] permits clients to divide databases into segments, which are somewhat related to Mneme’s
pools, though they do not support the same kind of application control over management of clusters. It also supports object-level clustering, which allows developers to control the placement of individual objects on disk pages and to control the number of objects that are stored on a particular disk page. Relational databases have always provided database administrator with means of (re)organizing the physical structuring of a database as well. Many existing clustering mechanisms are static, however—they allow objects to become members of clusters at object creation time. Reclustering is often possible using off-line database reorganization techniques. Our experiences, however, suggest that while static cluster definition is useful in many cases—in particular, when collections of objects most often are accessed together—dynamic, application-specific clustering is also important. For example, an application that performs breadth-first traversals of an abstract syntax graph would define clusters very differently from one that performs depth-first traversals. Further, some tools may not know \textit{a priori} how they will access objects (e.g., because their behavior depends on user input or on the results of interactions with other tools). In such cases, it would be helpful to be able to retain historical data about object accesses, and to use that data to guide future cluster definitions for those applications. Further exploration of this issue is deferred to future work.

Given that cluster definitions may be application-specific, the presence of concurrency may affect some of the signatures on operations specified in the object management framework. For example, it may be the case that the \texttt{Retrieve\_Persistent\_Object} operation would take an additional parameter, indicating an intended access pattern (e.g., breadth-first or depth first) or other clustering information.
CHAPTER 6
ARCHITECTURAL SUPPORT FOR CONCURRENcy CONTROL

Concurrency was not one of the original capabilities designed into Pleiades. Concurrency control has, however, a considerable impact on many aspects of the design of any object management system. Thus, one of the most significant aspects of adding concurrency control to Pleiades was to define and examine some alternative interface package architectures that would facilitate concurrent access.

Concurrency control comes at a cost (e.g., [15]). Achieving concurrency control requires the centralization of information about the kinds of concurrent accesses to objects that are occurring, which means that concurrent applications must communicate with this centralized information service via some form of interprocess communication (IPC). IPC is expensive. A major goal of our effort (as in any system that supports concurrent access), therefore, was to reduce the costs of concurrent access for clients that execute concurrently, while minimizing the impact of concurrency control on clients that do not execute concurrently. Much of our efforts were spent trying to determine what process boundaries were necessary, where to put them to minimize IPC costs while maximizing flexibility, and on evaluating the impact, with respect to a set of issues described below, of having the process boundary be in different locations.

In this chapter, we begin by introducing some issues that we had to address in considering different concurrency control support infrastructures for Pleiades. We then describe two alternative interface package architectures we have been exploring,
and we examine the relative advantages and disadvantages of each, based on how they address these issues.

6.1 General Issues

6.1.1 Optimistic vs. Pessimistic Protocols

One significant issue that must be addressed in any concurrency control manager is whether an optimistic or pessimistic concurrency control protocol will be used. Optimistic protocols [56] are based on various kinds of certification techniques. In general, optimistic protocols permit applications to access objects without restriction during a transaction. Upon an attempt to commit a transaction, a certifier can use one of several different algorithms to determine whether or not the transaction’s execution has violated serializability (or other correctness) requirements. If not, the transaction is allowed to commit; otherwise, it is aborted. Pessimistic protocols, on the other hand, are most often based on the use of locks. The most commonly used approach is strict two-phase locking (2PL) [37], which guarantees serializability and recoverability, but which is prone to deadlock. Timestamp ordering (TO) [120] is a non-locking alternative to 2PL. Like 2PL, TO guarantees serializability and recoverability, but unlike 2PL, TO also prevents deadlocks. Both 2PL and TO are limited in that they allow only a subset of the possible serializable schedules. 2PL does not allow schedules in which a transaction can obtain locks after it releases its first lock, while TO imposes an ordering on conflicting data accesses and causes the abort of any transaction that attempts to access a data item out of order. Thus, 2PL may require transactions to hold their locks for longer than the transactions actually need the locked data items, which reduces concurrency, while TO may cause the loss of significant amounts of work by aborting transactions whose data accesses come out of order. Serialization graph testing (SGT) [25] is a concurrency control mechanism that allows all serializable interleavings of data accesses. This flexibility comes, however,
at the cost of executing a cycle-detection algorithm on the serialization graph upon each data access from each transaction; this can make the technique prohibitively expensive. Strict 2PL, TO, and SGT all avoid cascading aborts, which means that the cost of preserving serializability is limited to the cost of aborting one transaction.

Optimistic and pessimistic concurrency control mechanisms are useful in different circumstances. Because they do not require the overhead of repeated (inter-process) communication with a concurrency control manager (e.g., to obtain and manage locks), they represent the least expensive mechanisms in circumstances where contention for objects is very low. Since optimistic mechanisms require the abort of all work done during the transaction, however, they can incur a high cost in terms of aborting a transaction that has otherwise run to completion, particularly if creative work was involved. They may also suffer from the problem of cascading aborts, which can be fairly serious.

Pessimistic mechanisms, on the other hand, precludes conflicting accesses. This makes them well-suited to circumstances when there is high contention for objects or when human effort is involved (i.e., when the cost of losing the work due to a conflict is unacceptable). Pessimistic mechanisms require more interprocess communication, however, so they may affect performance more negatively than optimistic mechanisms, unless enough work is done with retrieved objects to amortize the cost of the IPCs.

Complex applications may involve either automated or manual activities (or both). Thus, an object management system that supports them should allow both optimistic and pessimistic mechanisms to be used. A concurrent architecture for Pleiades should not mandate the use of either kind of protocol, in accordance with the generality and heterogeneity cross-cutting requirements.

Some objects are not accessed concurrently. For example, a compiler keeps a number of data structures that are for its own use; in a software development process, a manager may ensure that only one developer will work on a particular module at
any given time. When objects are not accessed concurrently, the applications that manipulate them should not have to pay for concurrency control. We would like to minimize (if not eliminate) the cost of concurrency control to such applications.

6.1.2 Granularity of Access

Objects may fall anywhere on a spectrum from fine- to coarse-grained. Files are examples of coarse-grained objects, while graphs are examples of fine-grained objects. The location of a process boundary can significantly affect the performance of an application that manipulates fine-grained objects vs. coarse-grained objects.

Another aspect of the granularity issue pertains to some pessimistic concurrency control mechanism. In a lock-based protocol, it is important to facilitate locking at different levels of granularity. When contention for objects is high, more concurrent access can occur if each application uses as fine-grained locks as possible. The more locks that are acquired, however, the higher the cost of concurrency control, and, in fact, Object Design, Inc. reported [79] that, despite a belief that pure object-level locking would improve concurrency, they found that in many cases, the performance penalty associated with the management of individual locks on large numbers of objects was too high. Thus, in cases where an application will be accessing collections of objects (e.g., an entire (sub)graph or a set), it is desirable to use coarse-grained locks over the whole (or large parts of the) collection. This reduces the number of locks acquired, which improves performance. An architecture for concurrent access should allow different granularities of locks to be acquired, and it should facilitate dynamic changing of lock granularity (as in, for example, the work performed by Carey et. al on adaptive lock granularity [24]). It should also support type-specific lock modes (e.g., semantics-based locking [9]). This flexibility is suggested by the cross-cutting requirements for heterogeneity and generality.
6.1.3 Evolvability

Applications, and the objects they manipulate, evolve over time. As new ADTs are defined, old ones are removed, and existing ones are modified, it is desirable to limit the impact of the changes on existing applications. In order of increasing impact, such changes can require a need to relink an executable or to change a dispatch table to use a new operation implementation, to recompile client code, or to recode part of the client. Insofar as possible, the impact on clients of changes to a concurrent version of Pleiades should be no greater than the impact of changes to the sequential version.

6.1.4 Support for Cooperative Development

Precluding non-serializable access to a shared object may be undesirable in certain circumstances. For example, teams of developers or collections of tools may need to cooperate with each other to produce or maintain a software product. In such cases, correctness criteria other than serializability must be used to ensure consistent access to objects.

In contexts where teams of people must cooperate to do their jobs, traditional concurrency control models, and architectures that support them, cannot be used to coordinate shared access to data or to facilitate cooperation [13]. A number of limitations have been pointed out; most notable among these is the fact that when developers cooperate, they may have to pass partial results back and forth in a way that is neither serializable nor atomic with respect to each other, but that may have to preserve serializability and atomicity with respect to other developers. With regard to architectural support, in a competitive situation, an application can keep the objects it manipulates in its own process space until it is finished with those objects, because no other applications can access those objects until it releases them. In a cooperative situation, however, an object update by one application may have to be propagated
to all the other applications with which it is cooperating. If support for cooperation is incorporated into an architecture that was intended for competitive access, the result may be very high interprocess communication costs—higher than might be needed using a different architecture. Similarly, an architecture that facilitates cooperative development may impose unnecessarily high overhead on applications that access objects competitively.

6.1.5 Orthogonality

Traditional database approaches to concurrency control have coupled several capabilities: atomicity, consistency, serializability, and persistence (otherwise known as the ACID properties). As has been noted elsewhere (e.g., [109, 26]), the coupling of these capabilities is often inappropriate. For example, it should be possible to achieve shared access (and thus, concurrency control) to a non-persistent object, and semantic consistency definitions may apply in both sequential and concurrent systems.

Over the past several years, a number of researchers have explored ways of relaxing the ACID properties to overcome some of these restrictions and thus, to better support software engineering activities. “Advanced” transaction models in general attempt to use type- or application-specific semantics to supplement or replace the traditional notion of serializability as the basis of concurrency control. Advanced transaction models represent different sets of trade-offs (e.g., flexibility at the cost of serializability), and thus, they provide different respective advantages and disadvantages. Predictably, no single transaction model has proven adequate for all, because every means of relaxing the ACID properties comes at some cost. Some costs are necessarily more acceptable to some applications than to others (e.g., an application that prints a cross-reference listing can be rerun without much pain if it aborts, but an application that supports design cannot be). Therefore, we believe that application developers must have the flexibility to decide which transaction features they must have, and which they are willing to sacrifice to obtain the semantics their applications
require. The choice of a transaction model will be based on application- and type-specific semantics and on end-user requirements, and the ability to choose or define an appropriate transaction model is mandated by the cross-cutting requirements for generality and heterogeneity.

Thus, we believe that satisfying the orthogonality requirement is particularly important, to ensure that different transaction models can be supported appropriately. An architecture for concurrency that requires the coupling of some or all of the ACID capabilities may significantly limit the ways in which the ACID properties can be relaxed.

6.2 ADT-Based Concurrency Control

One architectural style that we evaluated, shown in Figure 6.1, is based on concurrency control at the ADT level. This approach places a process boundary between an application and the PLEIADES-generated ADTs it uses. The approach is essentially
a “fat server” approach—it requires all object manipulation to occur at the ADT server, so applications incur one IPC per operation.\(^1\)

This architecture raises a fairly important issue. In particular, it requires a decision about whether to use a single ADT per server vs. multiple ADTs per server. At its extreme, where all ADTs are part of the same server, the multiple ADTs per server approach is essentially the architecture employed in traditional (non-distributed) databases. Use of a multi-ADT server minimizes the communication costs when one ADT refers to another. Its drawback is that the server may become a concurrency bottleneck. The single-ADT server improves concurrency at the cost of more than one IPC per operation (i.e., if an operation on one ADT invokes operations on others). It may be possible, in some cases, to maximize concurrency and minimize IPCs by partitioning ADTs into several multi-ADT servers.

With respect to the set of issues raised in Chapter 6.1, the ADT-based architecture has the following characteristics:

**Optimistic vs. pessimistic protocols:** ADT-based concurrency control is considerably more amenable to optimistic and lightweight locking protocols (i.e., where locks need not be acquired on all accessed objects) than to heavyweight locking, though this varies somewhat based on the particular architectural variant employed. The basic problem with heavyweight locking in this architecture is the performance. If clients have to acquire locks prior to invoking operations, they will take two IPCs per each operation that requires the acquisition of a lock, which is very costly. Except in the situation where all ADTs are bundled into the same server, an average cost of more than one IPC is also incurred if the ADTs acquire locks during the invocation of operations (the additional IPC overhead comes from the need to perform deadlock detection, which either requires a centralized wait-for graph, which adds an extra

\(^1\)Techniques exist that can, in some cases, reduce this cost. For example, it is often possible to ship multiple operation requests to the server simultaneously, which helps to amortize the IPC costs (as in, for example, Thor [60]).
IPC per lock acquired, or communication among ADT servers to detect deadlocks in a distributed manner). Placing all ADTs in a single server eliminates the need for additional interprocess communication, as the locking component can reside in the same server as the ADTs.

Optimistic protocols do not require much interprocess communication. This makes them well-suited to the ADT-based concurrency control architecture.

**Granularity of access:** With respect to object granularity, this architecture would perform poorly for fine-grained objects, since such objects are usually subject to multiple operations per execution of any tool that manipulates them. Given the IPC cost of each operation invocation, the overhead of this architecture for fine-grained objects is likely to be prohibitive. For coarse-grained objects, and particularly, for bulk objects, this architecture might perform quite well. For example, a query over a large set might return a small number of objects. Since the query runs at the server, the cost of shipping the whole set to the client is avoided.

Similarly, with respect to lock granularity, this architecture would not be expected to perform well for applications requiring fine-grained locks. It would likely perform better for applications using coarse-grained locks.

**Evolvability:** Different variants on the ADT-based architecture have different impact-of-change characteristics. On the single-ADT-per-server side of the spectrum, the impact of changing an existing ADT, or defining new ADTs or eliminating old ones, would be limited to the set of clients that actually use the affected ADTs. As multiple ADTs are bundled together into a single server, however, the impact of change may increase. For example, if $\text{ADT}_1$ and $\text{ADT}_2$ reside in a single server, a change to $\text{ADT}_1$ affects clients that use only $\text{ADT}_2$ (because the server changes when $\text{ADT}_1$ does). The benefit of this architecture with respect to evolvability is that it
ensures that all the applications that use a particular ADT will use the “right” version (i.e., because all clients of an ADT use the same ADT server).

**Support for cooperative development:** This architecture may be well-suited to cooperative development. It allows all objects to reside at the server, which means that any change effected by an application can become visible immediately to all cooperating applications—no additional communication costs are required to ship the modified object back to the server or to distribute it to the cooperating applications.

**Orthogonality:** This architecture imposes no orthogonality restrictions. All objects, persistent or transient, are manipulated at the server. Similarly, there is no visible difference between objects that are accessed concurrently vs. those that are not. Any object can potentially be accessed concurrently, independent of any other properties it may have (such as persistence).

The significant limitation of this architecture with respect to orthogonality is the performance penalty it imposes on manipulation of objects that are not subject to concurrent access. These objects are subject to the same IPC costs as shared objects, and there is little that can be done to reduce these costs. Thus, when not all of an ADT’s instances are subject to concurrent access, this architecture imposes too high a performance penalty.

### 6.3 Storage-Level Concurrency Control

The other architectural style we considered is shown in Figure 6.2. This architecture puts the process boundary between a Pleiades-generated ADT and the Storage Manager Interface (SMI), which was described in Chapter 5.2.1. Therefore, the majority of the ADT functionality is bound into each client application’s address space, while coordination among concurrently executing applications occurs at the level of persistent object manipulation. This is fairly different from the ADT-level
Fig. 6.2 Storage manager level concurrency control architecture.

architecture—objects reside at the application while they are being manipulated, which means that IPC costs are incurred only when persistent objects are requested from, and written to, the SMI. Locks are requested by the ADT. The lock manager component can be defined as a separate process, or it can be combined with the SMI, with different concomitant advantages and disadvantages (described below).

This architecture imposes at least a two-IPC cost per persistent object modified (one to retrieve the object from the SMI and one to commit the results of the update), and a one-IPC cost per object accessed without modification. It minimizes the costs of actually manipulating objects, however, since all other access to each object incurs only the cost of a local procedure call. Thus, the IPC costs are amortized in cases where a particular object is examined or modified repeatedly.

With respect to the set of issues raised in Chapter 6.1, the storage level concurrency control architecture has the following characteristics:

**Optimistic vs. pessimistic protocols:** This architecture is fairly neutral with respect to optimistic and pessimistic concurrency control mechanisms. For optimistic protocols, a validation manager replaces the lock manager shown in Figure
6.2. The validation manager is notified about object accesses (either incrementally or in batches), and the validation manager determines whether or not conflicting accesses have occurred upon transaction commit. For pessimistic mechanisms, any of the common protocols can be supported readily. The cost of lock-based protocols may vary, depending on where the lock manager resides. If it resides at the SMI, then locks can be acquired at the same time as a persistent object is accessed, which requires no additional IPC costs. (Additional IPCs are, of course, necessary if locks may be upgraded or downgraded.) If it resides separately, two additional IPCs are necessary per object updated (one to obtain a lock and one to release it). The advantage of the separate lock manager, however, is that it permits semantics-based locking to occur, which may improve concurrency. The SMI provides a very low-level object abstraction, so locking at the SMI level does not permit as much flexibility in locking.

**Granularity of access:** With respect to object granularity, this architecture would perform well for fine-grained objects; in fact, its performance improves with the number of accesses per object. It is not clear how well this architecture performs with respect to coarse-grained objects. If an object is accessed only a few times, the IPC costs to retrieve the object from the storage manager and then to send back any updates may be significant. Similarly, manipulation of bulk objects might be problematic. A query over a set, for example, might require a client to retrieve the whole set before performing the query. This may be costly, and it may reduce concurrency.

**Evolvability:** This architecture limits the impact of changing an ADT to the set of applications that use the ADT.

**Support for cooperative development:** This architecture does not facilitate cooperative development. In particular, since objects reside at the application while
the application manipulates them, cooperation requires additional IPC costs to ship updated objects back to the storage manager and then to the other cooperating applications. When “coarse-grained” cooperation is desired (e.g., allowing one author to write a section of a paper and then send it to another author for comments), this approach may be feasible. For finer-grained cooperation, however, the communication costs are likely to be untenable.

**Orthogonality:** This architecture facilitates concurrent access only to persistent objects. Clearly, this violates the orthogonality requirement. It is, of course, possible to simulate concurrent access to transient objects by making the transient objects persistent, but this comes at additional cost.

The advantage of this architecture with respect to orthogonality is that it imposes no additional performance penalties for the manipulation of non-shared, non-persistent objects. Such objects are created and manipulated at the client normally and need not be subject to the performance costs of the concurrency control mechanisms. The disadvantage of this architecture is that the coupling of concurrency control with persistence results in a performance penalty for accessing all persistent objects, even those that are not accessed concurrently.

### 6.4 Discussion

#### 6.4.1 Selecting an Architecture

Clearly, both architectures have their respective advantages and disadvantages in different contexts. The ADT-based architecture appears to be best for applications and objects with the following characteristics:

- A need for significant cooperative development.
• Low contention for objects. In particular, applications for which optimistic concurrency control mechanisms are appropriate may benefit from this architecture.

• Coarse-grained or bulk objects.

• Need for concurrent access to objects, independent of the objects’ persistence status.

This architecture is least suited in the following circumstances:

• Support for concurrent access to fine-grained objects, or repeated access to a given object.

• Support for “private” objects (i.e., objects that are not accessed concurrently).

In contrast, the storage-level architecture appears to be best for applications and objects with the following characteristics:

• Predominantly competitive access to objects.

• Fine-grained objects, objects that are accessed repeatedly, and fine-grained locking.

• Frequent simultaneous use of persistence and concurrency.

• Use of “private” objects.

• Optimistic or pessimistic concurrency control mechanisms.

This architecture is least suited in the following circumstances:

• Fine-grained cooperative development.

• Access to bulk ADTs (e.g., for queries).
• Concurrent access to non-persistent objects, and non-concurrent access to persistent objects.

It is evident that neither architecture is suited to all types of objects or applications. Instead of selecting one over the other, therefore, the approach we decided to pursue involved supporting both, and to use a combination of knowledge about typical access patterns over certain types of objects, developer suggestions, and historical data to guide the selection of an architecture for a particular ADT. With respect to knowledge about access patterns over types, most of the semantics of the extra type constructors PLEIADES supports—graphs, sequences, relationships, and relations—are defined by PLEIADES itself. We can, therefore, make reasonable guesses about appropriate architectural support for these types, as described below.

**Graphs:** Graphs tend to be fine-grained objects. Both nodes and edges within graphs can be, and usually are, attributed, which means that any object within a graph may be subject to examination or update by multiple operations each time an application accesses it. This suggests that the storage-level concurrency architecture is better suited to many types of graph objects.

**Sequences:** Sequences are bulk types whose members tend to be accessed in an all-or-nothing fashion (usually as part of an iteration loop). In our experience, it is uncommon for applications to access only some of the members of a sequence. Since sequences may be fairly large, the cost of one IPC per member accessed that the ADT-based architecture would impose seems prohibitively expensive. Therefore, we selected the storage-level architecture as the default for sequences.

**Relations:** Relations are somewhat problematic. They are bulk types whose members are accessed both associatively and navigationally (i.e., by iterating through the members of a relation). Associative access suggests the use of an ADT-based archi-
A combined ADT/storage-level concurrency control architecture.

Thus, except in cases where a developer can say that relations will, for the most part, be accessed associatively, support for concurrent access to relations may require the use of a mixed architecture, as depicted in Figure 6.3, with the process boundary occurring in the middle of the ADT. Using this architecture, associative accesses operations occur at the combined storage manager/ADT server. Navigational access operations, on the other hand, occur at the application/ADT side. This combined approach offers the benefits of both architectures in circumstances where neither alone is adequate. We note that this combined architecture is similar to that in a recent version of Mneme, in which Mneme permits multiple client/server splits within a single pool [76].
6.4.2 Implementation Issues

An implementation for any of the architectures described earlier in this chapter requires several infrastructure components. For lock-based pessimistic concurrency control, it requires a lock manager component. This module is responsible for the definition of lock modes and lock compatibility matrices, for granting and releasing locks, and for the bookkeeping associated with lock management. The lock manager is also responsible for enforcing any deadlock management policies. For optimistic concurrency control, a validation manager component is required (in place of the lock manager), which is used to determine whether or not one application’s set of object manipulations conflicted in any way with any other application’s.\(^2\) The storage-level concurrency architecture also requires an appropriate storage manager definition. For this, we used our existing Storage Manager Interface, which was described in Chapter 5.2.1.

For our initial experimentation, we decided to work exclusively with lock-based pessimistic concurrency control protocols. We made this decision because so many Pleiades client applications involve some amount of creative work, which makes the cost of failures due to concurrency conflicts too high. In addition, the effect on both architectures of the choice of optimistic vs. pessimistic concurrency control policies is minimal—an optimistic mechanism simply replaces the lock manager component with a validation manager. Thus, we did not believe that our decision to begin our experimentation with lock-based pessimistic concurrency control would adversely affect the applicability of our results.

We describe the infrastructure components briefly, and then discuss some issues that arose in implementing concurrency control in Pleiades.

\(^2\)Since optimistic concurrency control can be modeled as pessimistic control with breakable locks, it is actually possible to use a single lock manager component to support both optimistic and pessimistic control.
6.4.2.1 Infrastructure Components

The design of the lock manager component was constrained by our desire to (a) support semantics-based locks for those types of objects for which it is useful, (b) to facilitate any useful locking protocol (e.g., two-phase locking or altruistic locking [91]), and (c) to support the implementation of any useful deadlock management policy readily.

To satisfy the goal of supporting different types of locks, we defined a generic lock manager module [115]. The lock manager is instantiated with a set of lock modes and a lock compatibility matrix. The instantiated lock manager can then be used, either by collections of ADTs or by the storage manager, to acquire and release locks, and to determine various kinds of information about the locks that have been granted (e.g., which applications currently hold which locks, what applications are waiting for locks, etc.). As noted in [82], the lock manager is more comprehensive than those found in traditional database systems. It provides access to a variety of information about the state of the lock manager and about the status of clients and objects. This information is provided to facilitate the implementation of a variety of concurrency control models.

One issue we had to address in designing the lock manager component was whether to support dynamic control over the definition of the set of lock modes and the lock compatibility matrix. Although this form of control might be desirable at times (particularly in cases where on-line type evolution occurs, resulting in a type definition for which a new lock mode is needed, or where existing lock modes become subject to a different compatibility matrix), we decided against including it initially, because it was not clear to us how to handle changes to the set of lock modes and compatibility matrix when locks had already been granted, based on the original modes and compatibility. In particular, we could only envision allowing changes to occur during times when no locks are held, since a change that occurs while applications hold locks
means that the locks they hold, which did not conflict when the locks were granted, may either become conflicting or may cease to exist. Since it only seemed sensible to allow changes during quiescent times, we thought it was simpler and cleaner to support static control over lock modes and compatibility matrices. If dynamic control turns out to be important, and if appropriate semantics can be specified for it, the lock manager should be readily extensible.

To satisfy the goal of facilitating any useful locking protocol, we decided not to include knowledge of particular locking protocols in the lock manager. Instead, we incorporated this knowledge into the Pleiades ADTs. This means, for example, that the two-phase locking protocol is accomplished by having Pleiades ADTs enforce a policy that no new locks can be acquired after a lock has been released.

Satisfying the goal of facilitating any useful deadlock management policy meant that we had to provide some means of determining when a deadlock had occurred (to support deadlock detection) or could occur (to support deadlock avoidance). Therefore, we developed a waits-for graph module to make this information available. The lock manager can optionally inform the waits-for graph module when applications are waiting for locks currently held by other applications. Communication with the waits-for graph module is done optionally because some concurrency control models either do not allow clients to wait for locks that cannot be granted immediately, or they can determine in advance that deadlocks will not occur (i.e., using deadlock prevention techniques); such models are not subject to deadlock, so they do not require the overhead of maintaining a waits-for graph. The appropriate use of the lock manager and waits-for graph modules readily facilitates the use of any deadlock management policy.

6.4.2.2 Experimental Results

To date, we have completed some simple, preliminary experiments with the storage-level architecture; experimentation with the ADT-based architecture is deferred.
to future work, though we note that one client ([108]), which is discussed in more
detail in Chapter 8, has already implemented this architecture manually for their
PLEIADES ADTs. We serverized the Storage Manager Interface (SMI) using the Q
[68] interprocess communication mechanism. We then manually modified an existing
PLEIADES ADT—IRIS [10], which is a language-independent abstract syntax graph
that is at the core of the Arcadia language processing and analysis tools—to support
two-phase locking and to use the lock manager. The modified version of IRIS was
linked into some existing IRIS-based tools, and these tools were used to evaluate the
impact of adding concurrency control.

The results of our experimentation, which are discussed in detail in [82], were
fairly encouraging. The cost of retrieving objects from a persistent store increased
by a factor of two, while the cost of writing objects increased by a factor of 2.7.
The increased time includes the cost of acquiring locks, maintaining the waits-for
graph, and of the IPCs now involved in communicating with the SMI. Beyond this,
the performance of the IRIS manipulation operations did not change appreciably.
We are encouraged by these preliminary results because the experimentation helped
us to determine that some of the SMI functionality was poorly designed for use in
the storage-level concurrency control architecture. In particular, some object validity
checks were occurring via operation calls to the SMI made from the PLEIADES inter-
face packages. While this did not cause any difficulties when the SMI was bound into
the same process space as the PLEIADES ADTs, it became more problematic when
every procedure call to the SMI incurred the cost of an IPC. Moving the checks to
the SMI will significantly reduce the number of IPCs required, which will reduce the
performance costs associated with object retrieval.

As noted in Chapter 5.2.2, the definition of related clusters of objects became
an important one in supporting the storage-level concurrency architecture, because
retrieval of objects from the SMI requires an IPC, so reducing this cost is impor-
tant. Experimentation with clustering strategies was deferred to future work, but we incorporated various “hooks” in the SMI to support the retrieval of clusters of objects.

6.5 Other Issues

Some design-level issues arose in our work on architectural support for concurrency control. These are discussed below.

**Achieving atomicity and serializability:** We have used an approach to concurrency control which, like our approach to persistence (Chapter 5), places much of the burden for enforcing concurrency control policies on the individual ADTs. For concurrency control alone, this is not problematic. Transactions, which are the unit of concurrency control in database systems, however, also incorporate notions of atomicity and serializability that are often desirable properties. Atomicity and serializability are, however, more “global” properties—they span concurrent access to multiple ADTs. Thus, to achieve these properties, it is necessary to introduce an entity, like the database transaction, that delineates the scope of object manipulations that are to be treated as atomic and/or serializable. This entity corresponds to type `Transaction`, described in Chapter 3.5.2. It provides the centralized control that is needed to coordinate the individual concurrency control actions of each ADT and to enforce such global properties as atomicity and serializability. It may also help to enforce policies such as two-phase locking and various forms of deadlock management.

As described in Chapter 5.2.3, we chose to implement the `Transaction` type as part of the repository manager abstraction. The initial implementation of this type did not conform to the specification presented in Chapter 3.5.2. In particular, transactions couple support for serializability, atomicity, and persistence, so the transaction-related operations do not accept sets of desired transaction properties as parameters. This
was done mainly as an expedient, since the initial implementation did not support concurrent access. Our ongoing work in adding concurrency control to Pleiades will fully support the specification of type Transaction.

**Orthogonality:** In general, database systems acquire locks on objects implicitly, as part of the implementations of the operations that manipulate those objects—a lock abstraction is not presented to applications directly. Each operation acquires the weakest lock possible for its purposes, which helps to improve concurrent access. This model is appealing, because it means that the particulars of the underlying concurrency control protocols are not coupled with ADT abstractions, which helps make concurrency control orthogonal from other object management capabilities.

The main disadvantage we found in retaining complete orthogonality is that implicit locking may result in additional IPC costs if a lock on an object has to be upgraded later. For example, an operation that returns the value of a node’s attribute would acquire a read lock. If the application manipulating that node later needs to update the attribute’s value, it would have to take the additional performance cost of contacting the lock manager component to upgrade the lock. In cases where the application knows the strongest lock it will ultimately need, the cost of upgrading locks may be unacceptable, though acquiring the strongest lock immediately may result in reduced concurrency. The problem may be even worse in cases where the application and/or human user performs a considerable amount of work before attempting to upgrade the lock, since it may not be possible to acquire the new lock at the point it is needed, which would result in wasted effort (and a cranky user). Thus, it may be desirable to allow clients to request stronger locks than those that would be acquired by default. Providing this control, however, requires a violation of orthogonality: either the signatures of all operations on an ADT must change to incorporate a “desired lock mode” parameter, or new operations must be defined to
allow clients to request and release locks explicitly. We note that supporting some kinds of advanced concurrency control models may require an orthogonality violation in any case. For example, altruistic locking [91] is a model in which clients release locks on objects as soon as they are finished with those objects. Although it may be possible, via various data flow analysis techniques, to determine automatically when an application is finished with some objects, automatic determination is not possible in all cases, which means that applications may have to indicate when they are finished using objects. This suggests that complete orthogonality may not be possible.
CHAPTER 7
INSTANTIATING THE CONSISTENCY MANAGEMENT MODEL

Producing a mapping from the framework described in Chapter 3 to Ada required us to address several design and implementation issues. We describe some of these issues in this chapter.

7.1 Design-Level Issues in Mapping the Framework to Ada

Satisfying the cross-cutting requirements in a consistency management model proved to be difficult in some cases. In other cases, satisfying the cross-cutting requirements had some interesting implications for our object management framework. These issues are discussed below.

First-class status and identity of constraints and actions: The framework described in Chapter 3 modeled constraints and actions as special kinds of operations, and operations are first-class objects. In Ada, however, operations are not first-class entities. The behavioral semantics of constraints and actions are such that we decided to model them as Ada operations anyway. As it turned out, the effect of this decision was profound, because we relied on the first-class status and identity of constraints and actions to enable us to model the definition and enforcement of relationships and constraints among constraints and/or actions. For example, it is often important to enforce relationships among constraints and/or actions, such as “Is_Acyclic should be checked on a graph before Passes_Analysis,” “Is_Releasable is
decomposed into Compiles, Links, and Passed_Review” and “upon violation of the up-to-datedness constraint between a CFG and its corresponding ASG, first run the action to try to recompute the CFG, and then run the action to notify all affected developers about whether or not the computation succeeded.” By choosing to map constraints and actions into Ada operations, we were left without the ability to model these kinds of interrelationships readily. As will be discussed in Chapter 8, this limitation proved to be a very serious one for some Pleiades clients.

Our experience has led us to conclude that, for mapping the framework to programming languages in which operations are not first-class entities, it is better to map constraints and actions to first-class objects, and then provide appropriate manipulation operations on those objects. Basically, this means that the operation Invoke must be defined explicitly on constraint and action objects (whereas it comes for free with operations). This model is not as clean from a client’s perspective, since the client must differentiate syntactically between invoking an operation and invoking an action or checking a constraint, but it would have provided better modeling power.

**Computational completeness of constraints:** This requirement ensures that any consistency definition that is necessary can be specified. Computationally complete formalisms, however, are very difficult to reason about. Reasoning about consistency definitions in particular would be a very useful capability—for example, to be able to decide whether two constraints are mutually unsatisfiable, or to be able to determine automatically the set of operations in which a given constraint can be violated.

**Dynamic control over enforcement of constraints:** Satisfying the dynamic control requirement was not, of itself, difficult. Doing so, however, ended up having some interesting implications for the specification of constraints. Consider, for example, the specification of a uniqueness constraint defined on a set of names. If this
constraint is enforced during times when it could be violated, it might be specified in such a way as to be checkable incrementally (i.e., upon adding a new name to the set). Let $New\_Name$ be a name that is about to be added to the set $Name\_Set$. The uniqueness constraint might be specified abstractly as:

$$\begin{align*}
\text{if } ( \text{not } New\_Name \in Name\_Set ) \text{ then} \\
\quad \text{-- constraint satisfied} \\
\text{else} \\
\quad \text{-- constraint violated; do not allow addition} \\
\text{end if;}
\end{align*}$$

If, however, the constraint is not enforced at all times while updates can occur, then, upon enforcement of the constraint, the following check must be performed:

$$\begin{align*}
\forall Name_1, Name_2 \in Name\_Set, \\
\quad \text{if } Name_1 = Name_2 \text{ then} \\
\quad \quad \text{-- constraint violated} \\
\quad \text{end if;}
\end{align*}$$

Although the second specification can be used to check the uniqueness constraint incrementally, the first one is preferable as an incremental check because its complexity is, at worst, $O(n)$ in the number of names in the set (and, with the use of indices, it would probably be $O(\log n)$, or, if hashing can be used, $O(1)$), while the second check’s complexity is $O(n^2)$ (or $O(n)$ with hashing).

Thus, in the presence of dynamic control over enforcement and relaxation of constraints, it may be desirable to support different specifications of a given constraint—ones that are used to check for the satisfaction of the constraint at enforcement-time, and others that are used as incremental tests. This type of control would manifest itself in a change to the signature of the $Enforce$ operation defined in Chapter 3.4.2:

```
procedure Enforce ( C : Constraint; 
\quad On\_Object : MetaType; 
\quad Actions : set of Action; 
\quad Check\_As : set of <Operation, When>; 
\quad Enforce\_Time\_Check : <Constraint, set of Action> );
```

The new parameter, $Enforce\_Time\_Check$, optionally specifies an alternative constraint specification and set of actions to be used at enforcement-time only.
We note that ideally, application developers would be able to specify just one form of constraint, and then leave the object management system to produce incrementally checkable constraints. This type of transformation cannot be deduced automatically in the general case, however. The presence of a computationally complete constraint specification formalism complicates the problem further.

### 7.2 Implementation-Level Issues

The implementation of the consistency management part of the object management framework proved to be fairly complicated. This was due primarily to a problem that represents a low-level interaction between the cross-cutting requirement for object identity and the consistency management model. We describe this problem, and its implications on consistency management, in this section.

To discuss this interaction issue, we will use an example, depicted in Figure 7.1, taken from the world-wide web. In a given organization, each employee has his/her own home page. Each page contains the employee’s name, represented as a reference to a name object. The organization itself has a home page, which contains a directory of the names of all employees and links to their home pages. The company enforces
a uniqueness constraint on the directory—no two employee names can be exactly the same—to facilitate the location of any given employee.

Initially, the company has two employees: Joe Brown and Joe Smith. At some point, Joe Brown marries Molly Smith and changes his name to Joe Smith. Upon returning to work after his honeymoon, he changes the name object to which his home page refers. With this change, however, he violates the unique names constraint defined on the company’s home page directory. Further, he does so without ever directly modifying the directory itself—he simply modified an object to which the directory refers, and on whose state the directory depends.

We refer to this problem as the container problem, because it arises when the state of one object depends, either directly or indirectly (transitively), on the states of other objects which, in some sense, the object contains. This problem is found commonly in software systems. Perhaps the best-known example of this problem is the dangling reference problem, which occurs when an application destroys an object to which other objects still refer (resulting in a violation of the referential integrity constraint). The container problem has at least three necessary conditions. First, it can only occur in the presence of identity, where “identity” means that an object has some way of referring to it that is independent of its value or state. The container problem does not occur in purely value-based models. Second, it can occur when the state of an object, $o_1$, depends, in some way, on the state of another (contained) object, $o_2$. Third, it can occur when a contained object can be manipulated in such a way as to violate a consistency definition on any object that contains it. For example, the dangling reference problem cannot occur if objects do not provide a Destroy operation.

The container problem has some very serious implications for implementing consistency management. For incrementally checked constraints, it is not sufficient to check the constraint solely upon access to the object that to which the constraint
applies (e.g., the directory object in Figure 7.1). Instead, it is necessary to identify the full set of objects whose state affects a given object’s consistency, determine what manipulations of those objects could potentially violate a consistency condition, and check the constraint at any point when it could be violated. This is a very difficult problem, for a number of reasons. The set of “contained” objects may be very large, given the transitive definition of containment. Further, this set of objects may change throughout the lifetime of a container object. Thus, identifying the set of contained objects, and performing the checks required to maintain consistency, may be very costly. Clearly, in the presence of distribution and concurrent access, it may not even be possible [109].

Numerous ad-hoc solutions have been taken to try to address the container problem, and many of them work by eliminating one of the necessary conditions for the problem to occur. In the memory management community, garbage collection has been used to address the dangling reference problem. This approach removes applications’ ability to destroy objects explicitly—in instead, the memory management system determines when objects become unreachable, and, at that time, it discards them. This eliminates the third necessary condition for the container problem to occur. While garbage collection techniques typically scale well, their major problem is that they depend on a static property of “garbage”—i.e., once an object becomes garbage, it remains garbage. Thus, garbage collection can occur at any point after an object becomes unreferenced. For many kinds of consistency conditions other than referential integrity, however, this kind of static property does not exist—changes in constrained objects occur frequently, which limits the utility of such approaches. *Invertible pointers* are used fairly pervasively in object-oriented database systems [8] to help address the container problem. In this approach, pointers are essentially bidirectional; thus, “contained” objects would refer back to their “containers.” In some cases, this technique works well, since contained objects can then cooperate with their
containers to enforce consistency constraints. The cost of this approach is primarily in terms of space, since each reference actually requires two references (forward and backward). This cost may be unacceptable, particularly in cases where objects contain numerous references (as in, for example, many types of graphs). This approach also does not scale well, and it increases coupling among objects, since contained objects are required to know about their containers. Further, invertible pointers may have associated costs in terms of performance and reduced concurrency. For example, if an object $o_1$ initially refers to $o_2$ and is updated to refer to $o_3$ instead, the reverse pointers from both $o_2$ and $o_3$ must be updated appropriately. The additional updates take time, and they require update locks on all three objects, instead of just an update lock on $o_1$ and read locks on $o_2$ and $o_3$, which reduces concurrency; further, the existence of any lock on $o_2$ will preclude the modification to $o_1$. Event-based notification techniques (e.g., [85]) represent another approach to addressing the container problem. These approaches require objects to announce significant events to a message server. Any objects that are interested in these events register to receive the events. The information made available through broadcast events can then be used to determine when to check particular consistency conditions. The major benefit to this approach is that it does not require "contained" objects to know about their containers. Its disadvantage is that it does not scale well in the presence of fine-grained or often-accessed objects, since in both cases, large numbers of events are announced. Further, event-based notification techniques do not solve the problem of determining which objects affect a given object—they leave this problem to consistency management systems. Finally, lazy evaluation techniques, such as polling, are approaches that can be used in some cases. Enforced constraints are checked periodically, rather than immediately upon an action that represents a potential consistency violation. The only guarantee that is made is that consistency violations will be detected prior to the next access to an inconsistent objects. This approach scales well, and it is
useful in situations where applications only need to know that they will not be permitted to access inconsistent objects. It is not, however, powerful enough to enforce all necessary kinds of consistency conditions. In particular, while it may work well for many kinds of “roll-forward” semantics (i.e., a repair action is taken upon detection of a violation), it cannot preclude erroneous forms of object manipulation. In the example shown in Figure 7.1, for instance, polling could not be used to enforce the unique names constraint. Attempting to use lazy approaches in such circumstances imposes additional restrictions on the underlying implementation of an object—for example, it would require the directory object’s index structure to permit duplicates, a capability that comes at additional cost.

It is clear that, while all of these approaches are useful in some circumstances, none of them scales in all cases. This suggests that implementing consistency management requires supporting multiple strategies. It also suggests that application developers will have to provide additional information, at least in some cases, to help the object management system choose an appropriate implementation mechanism. This information may include:

- intended enforcement semantics (e.g., whether or not violation is precluded).
- explicit delineation of the periods of time during which it is acceptable for an enforced constraint to be violated. This requires differentiating between relaxing a constraint and temporarily violating a constraint.
- the set of contained objects, and an indication of how manipulation of contained objects can affect constraints defined on a container.
- other information useful to the selection of an implementation mechanism, such as the presence of invertible pointers as part of the definition of an object.

Clearly, allowing developers to provide at least some of this information would require changes to the definition of type Constraint in the object management framework.
Various kinds of analysis techniques may be very useful in helping to guide the selection of an implementation mechanism. For example, dataflow analysis could help determine the set of operations that might modify a given object, thus violating constraints on the object. We defer further exploration of the application of analysis technology, and the use of developer-supplied additional information, to future work.

Finally, we note that the container problem is significantly complicated by the heterogeneity cross-cutting requirement. The problem arises whenever an incrementally checked consistency condition must be defined across objects specified using different languages or object management systems, because accesses to objects defined in other systems cannot natively be detected by an external consistency management system. Only notification-like solutions can be used to address this particular kind of problem (i.e., where one object management system notifies the consistency manager of another object management system when relevant events occur).

7.3 Interaction Issues

Achieving orthogonality proved to be somewhat difficult, since consistency management interacts with some other kinds of object management capabilities. We describe some of these interactions in this section.

7.3.1 Consistency Management and Persistence

The ability to exert dynamic control over object persistence has an interesting interaction with consistency management. Objects that satisfy their consistency constraints may, in fact, cease to satisfy them when the objects become persistent. This is because the consistency constraints may depend on the states of any objects reachable from a constrained object (i.e., “contained” objects), but not all contained objects need become persistent (because of the instance-based control over the extent of persistence). Thus, it is possible to define conflicting consistency and persistence models,
and at least some of these conflicts cannot be detected until run-time (i.e., in cases where persistence decisions are made dynamically).

There are a few possible solutions to this problem. One is to check the persistence status of all the objects a “container” object contains (transitively) as soon as the “container” becomes persistent, to ensure that the container’s consistency definition(s) will not be violated. Subsequently, each update to a “contained” object, including a change in its persistence status, must result in rechecking the container’s consistency constraints. This approach may be extremely expensive, however. Another solution, which is the one taken in most database systems, involves checking all constraints at the end of a block of code that manipulates a set of objects (in database systems, it would occur at the end of a transaction). If the set of persistent objects do not satisfy their consistency definitions, the block is aborted. This approach is not feasible in cases where failure to detect violations of consistency definitions in a timely fashion may result in significant amounts of invalid work, but it is useful in cases where it is acceptable to check a constraint as a postcondition to a collection of activities.

We note that no matter what approach is used, this interaction limits the degree to which persistence and consistency can be made independent capabilities. It is not possible to achieve complete orthogonality of consistency management and persistence, because the persistence status of an object may affect its consistency status.

7.3.2 Types, Persistence, and Consistency Management

The ability to control the enforcement of consistency conditions at both the instance and type level is an important capability. Type-level enforcement can be achieved, based on the model in the object management framework, in two ways: by enforcing a constraint on the type itself, or by enforcing the constraint on each object in the Extent_Of the type. Unfortunately, an Ada instantiation of the framework does not readily facilitate the enforcement of constraints on types, because types are not
first-class objects in Ada. This leaves only the second mechanism: using the Extent_Of operation.

Chapter 5.3.1 described an interaction between persistence and the definition of the Extent_Of operation. Essentially, the problem is that different definitions of “extent” may be appropriate for different applications. In some cases, applications may want “extent” to mean “all existing instances of a type,” while in other cases, “extent” may be relative to the space of objects that are visible to a given application (e.g., the set of instances of a type in a given repository). This interaction further conflicts with the ability to enforce type-level consistency definitions.

7.3.3 Concurrency Control and Consistency Management

Numerous objects may affect the consistency of any given object. This suggests that, while a particular application may directly manipulate only a small number of objects, it may need access to a much larger set of objects for purposes of evaluating and enforcing consistency definitions [107].

In the presence of concurrent access, it may not be possible for an application to obtain access to the set of “auxiliary” objects needed to evaluate the applicable set of constraints. (This is, in fact, part of why the operation Satisfied, which is defined on type Constraint, can return the value Unknown.) In fact, this problem has ramifications for concurrency control and deadlock management policies that rely on knowing the set of objects that an application will access (e.g., for deadlock prevention algorithms and for concurrency control mechanisms that schedule transactions based on when objects will be available). In the presence of consistency management, the set of objects that affect an application can be a superset of the set of objects the application explicitly accesses, and that set may not be determinable statically.

The ramifications of this interaction between concurrency control and consistency management on applications may be profound. It is possible for applications to acquire access (conservatively) to all the objects they require, perform their tasks
(perhaps with human input), and find that either their activities violated a consistency definition, or that they are unable to obtain access to objects whose states affect relevant consistency definitions. This has implications for concurrency control models, which may have to take into account factors such as priorities and relative importance of activities in deciding how to resolve concurrency conflicts. It also has implications on consistency management mechanisms, which, in addition to supporting the specification of properties like priorities and relative importance, may have to facilitate the identification of the set of objects that affect a given consistency condition to allow, for example, applications to acquire locks on the full set of objects that will affect it.
CHAPTER 8
EVALUATION

PLEIADES is currently in use in several real-world applications, both academic and industrial. We have used some existing clients to help us evaluate the PLEIADES prototype, the object management framework, and our underlying hypotheses about object management support for complex applications. This chapter presents our evaluations. Each section gives an overview of a particular client application, and then discusses how each of the five kinds of PLEIADES functionalities (i.e., type model, navigational and associative access, persistence, consistency management, and concurrency control) were used to support the definition of the client. We focus particularly on what clients found to be the strengths and limitations of PLEIADES, and how those strengths and limitations relate to the functionality dictated by the cross-cutting requirements, in an effort to evaluate whether or not this functionality really facilitates the construction of complex applications.

We note that the evaluations presented in this section were compiled based on information and feedback obtained directly from several PLEIADES users [2, 46, 69, 81, 108, 129] in an attempt to minimize the subjectiveness inherent in such an evaluation. The process we used in producing these evaluations was as follows. We constructed a questionnaire that included approximately fifty questions. The questions were divided into six sections: general questions about overall usability, followed by questions about each of the areas of functionality PLEIADES supports (type model, navigational and associative access, persistence, consistency management, and concurrency control). In each area, the questions attempted to determine whether, and how, each PLEIADES
client had used functionality dictated by the cross-cutting requirements, how closely the functionality provided matched the client’s needs, and whether current limitations in PLEIADES (due either to Ada limitations or to PLEIADES-specific restrictions), or any other aspect of PLEIADES’ semantics, caused the client difficulties. Once a client had returned the questionnaire, we followed up with additional questions to clarify some of the responses. We then performed an evaluation of that client’s experiences, based on the information provided. Once this evaluation was written, it was sent to the client for feedback. All evaluations presented in this section have been approved by the appropriate clients.

8.1 Reusable Components Library

The LASER components library [129] is a collection of reusable abstract data types, written in Ada. The library includes standard generic data types, like lists, several kinds of sets, stacks, balanced binary trees, and symbol tables, along with some useful utility data types, such as a command-line interface, various calendar utilities, and string/character manipulation packages.

Most of the modules in the reusable components library were developed using PLEIADES, but they share a common goal: to be usable in any Ada program. For the generic data types, this also means that the types should be instantiable on any Ada ADTs, whether or not those ADTs were defined using PLEIADES.

**Type model:** Most of the reusable components mapped directly or readily to PLEIADES type constructors. For example, stacks were modeled using sequences, binary trees were implemented using PLEIADES graphs, and relations were used to model general-purpose sets. Two major type model limitations were encountered in the development of some reusable components. One was the PLEIADES limitation that relations can only contain relationships. This was problematic in the implementation of sets, because it required the definition of a single-field relationship to hold set
elements, rather than allowing the set elements to be placed directly into the relation. The second type model limitation was discovered during the implementation of symbol tables, and it was a little more serious. In particular, symbol tables are an example of a data structure over which both associative and ordered navigational accesses occur frequently. Associative accesses occur when applications look up the definition of a given identifier in a specified context. Ordered iteration occurs when applications examine all of the definitions that occurred in a particular scope. PLEIADES relations do not support ordered iteration readily, while sequences do not support queries. To obtain the required functionality, the developer of the symbol table abstract data type had to maintain dual representations of each scope—a sequence and a relation—and keep them mutually consistent. Clearly, this is not a desirable solution. This experience pointed up the need for an indexable, queryable type constructor.

**Navigational and associative access:** The reusable components are a fairly mixed bunch with respect to navigational and associative access. Some of the components, like lists, are accessed exclusively via navigation, while others, like sets, are accessed predominantly via associative access. Some components, like symbol tables and binary trees, are accessed both navigationally and associatively.

While the reusable components library developer found PLEIADES’ support for navigational and associative access to be appropriate, he noted that the inability to support both navigational and associative access to the same data structures was a limitation, as described above. This limitation manifested itself in the need for queries over sequences, and also in the need for queries over graphs.

**Persistence:** All of the reusable components provide dynamic, instance-based application control over the persistence of ADT instances. The interface to the persistence mechanism is via the standard PLEIADES Get_PID and Get_OID operations (Section 5.1.2). For the generic components, instantiators are also required to supply
Get_PID and Get_OID operations for the type on which the generic is instantiated (e.g., for the stack element type in the stack ADT).

With these operations, the generic components are able to become persistent, and to retain persistent references to their elements, irrespective of the particular underlying persistent object system in which the elements are stored.

Some users of the reusable components library noted that, while they did require instance-based control over persistence, they would have liked to have been able to change the default persistence of objects. For example, one developer indicated that most of the binary trees he created would, in the normal course of events, become persistent, so he would have preferred to have binary trees be persistent by default, and then be able to indicate which trees should not become persistent. Although it is possible to achieve these semantics using PLEIADES, this feedback suggested that explicit control over the default value of an object’s persistence attribute should be available.

Some of the reusable components made use of the persistence operations to supply specially optimized in-memory representations of objects. For example, one implementation of the sets ADT manipulates a bit vector representation to optimize the performance of certain classes of applications. The persistent representation of these sets, however, is the same as for other set implementations—several of the alternative set implementations share this representation, which allows different applications to manipulate the same (persistent) sets using different in-memory representations that are optimized for different access characteristics. As it turned out, having Get_PID and Get_OID as the interface to the persistence mechanism allowed the sets ADT developer with the necessary “hooks” to facilitate the dual in-memory/persistent representations. The developer simply provided his own implementations for these operations, where Get_OID retrieves a persistent set and copies it into an optimized

---

The definition of these operations is specified in more detail in [112].
representation, and, at the point where modified persistent objects are written back to the persistent store, the optimized representation is copied back into the persistent version; this is essentially the copy-swizzle approach described in [75].

**Consistency management:** Various reusable components are subject to certain kinds of consistency definitions. These definitions can be divided into two categories: *semantic integrity constraints*, which restrict the sets of states objects can take, and *operation constraints*, which indicate invocation relationships among operations on ADTs. An example of a semantic integrity constraint is found in some kinds of sets, which impose uniqueness constraints (i.e., no two members may have the same value). Stacks are examples of ADTs with operation constraints. For example, the Pop operation may not be executed on a given stack unless at least one more Push than Pop has occurred on the stack.

The reusable components library provides good examples of ADTs that are subject to fairly different kinds of consistency models, and that may be subject to different models based on which application is using the ADTs. For example, one client who instantiates the sets generic needed all instances of sets to enforce the member uniqueness constraint, while another does not use the uniqueness constraint at all, and another imposes the uniqueness constraint on some sets, but not all. This underscores the importance of having control over constraint enforcement at both the type and instance level, and of being able to make decisions dynamically about the set of instances that have a particular constraint imposed. The need for operation constraints also suggests the importance of modeling operations as first-class objects, to facilitate the description of relationships among, and constraints over, operations. Because Pleiades is an extension of Ada, which does not make operations first-class entities, it was very difficult to enforce operation constraints using the consistency
management mechanism. Thus, the components library developers had to implement such constraints manually, as part of the implementations of operations.

**Concurrency control:** We found two predominant means of accessing reusable components among the clients of these components. In some cases, clients used the components to create objects that are essentially “private”—i.e., only a single client uses these objects. An example of this kind of client is Ada_To_IRIS (described later in this section), which translates Ada source code into the IRIS abstract syntax graph representation. Ada_To_IRIS creates several data structures, like parse stacks, that are intended for its own use. In such cases, the cost of the mechanisms that permit concurrent access is both unnecessary and undesirable. In other cases, clients create instances of reusable components that are shared among applications. For example, the FLAVERS toolset (described in Section 8.3) analyzes programs to verify properties of those programs. The toolset operates on a trace flow graph (TFG), which is a representation of concurrent programs that shows (potential) flow of control through programs [35]. Each node in the TFG is annotated with a set, which contains the collection of states in which the node (representing a statement in the program) can be visited. Different tools may use this information as part of their analysis of a program. Concurrent access to these sets is, therefore, required.

Different concurrent architectures may be appropriate for applications that access different ADTs. For example, stacks are manipulated exclusively through single-element accessor operations—that is, **Push** adds one element to the stack, **Pop** removes one element from the stack, and **Top** examines the element at the top of the stack. No iterative access is permitted, and no bulk accessor operations (e.g., queries) are supported. If both **Pop** and **Push** are executed often by a single application (as is often the case), the storage-level concurrency control architecture may perform considerably better than the ADT-level architecture. In cases where a reader/writer
situation exists (i.e., one application adds elements to the stack while another removes them), the ADT-level architecture is clearly preferable.

8.2 Language Processing Toolset

The Arcadia language processing toolset [127, 128] comprises a set of tools that produce and maintain various interrelated representations of source code. All of these representations, which are described below, were developed using Pleiades. In this section, we present some of the representations and tools, then discuss how each of the Pleiades functionalities were used to support their definition.

8.2.1 IRIS

IRIS is a language-independent abstract syntax graph ADT. IRIS graphs have two kinds of nodes: reference nodes, which represent the use of an identifier or literal, and application nodes, which represent the application of an operator to zero or more operands (in a very LISP-like style). The number of operands associated with a given operator can change at any point. The operator itself is represented as the zero-th operand of each application node. IRIS distinguishes five different varieties of reference nodes, corresponding to different kinds of literals: string literal, character literal, integer literal, float literal, fixed-point literal, identifier, and resolved token. Resolved tokens are used to represent a reference to an entity that is declared elsewhere. Such references point back to the full declaration of the referenced entity. Figure 8.1 shows, as an example, a representation for the assignment statement “X := 5;” in IRIS. The assignment statement itself represents the application of the operator := to the operands X and 5, so it is represented as an application node. 5 is represented by a reference node whose kind is integer literal. X is a use of a variable declared elsewhere, so it is represented as a reference node whose kind is resolved token, and it points to the declaration of the variable X.
Figure 8.1 IRIS representation for an assignment statement.
By definition, IRIS graphs are always trees, except where resolved tokens occur in the graph. Thus, each node has at most one parent. This property greatly facilitates the processing of IRIS graphs, and there are no circumstances under which it is correct to violate this property. It is always possible to navigate from any node in an IRIS graph to its parent—thus, edges in the graph are bidirectional.

**Type model:** The definition of IRIS in Pleiades was fairly straightforward. It is defined as a graph with two types of nodes, application and reference. Application nodes contain an operator attribute and a sequence of IRIS nodes (application, reference, or both), corresponding to the set of operands to which the operator is applied. Since the operator is actually the zero-th operand, the operator attribute is simply an alias for the zero-th operand. Sequences were well-suited to modeling the operands of an application node, given that an application node’s number of operands can change dynamically.

Only one type-level issue arose in the definition of IRIS. The elements of a sequence are always numbered $1..\text{Length(Sequence)}$. The operands of an application node, however, are numbered starting at zero (since the operator is represented as operand zero). The lack of control over the index type proved to be a minor nuisance, since the implementation of the IRIS operand accessor operations had to adjust the operand numbers accordingly. This incurred extra run-time overhead, and it also reduced code comprehensibility.

**Navigational and associative access:** IRIS graphs are accessed almost exclusively via navigational access from the root. For the most part, the interpretation of any node in an IRIS graph depends on the context in which it appears, which significantly reduces the types of queries that can be asked of an IRIS graph itself. For example, tools might want to know the locations of all references to the variable
X in a particular scope. Scopes, however, are language-specific concepts; thus, they are not part of the IRIS abstraction.

**Persistence:** The natural persistence model for IRIS graphs is navigational reachability, and this is the model incorporated. The majority of IRIS objects that are created do eventually become persistent, which led the IRIS developers to consider a persistence-by-type model. Some tools do, however, create IRIS graph fragments as intermediate results and later discard them, so the instance-based, dynamic control turned out to be useful.

**Consistency management:** As described above, IRIS graphs are subject to a well-formedness constraint: nodes have at most one parent. This constraint is subject to preclusion enforcement semantics—it is never correct to violate the constraint. Thus, the constraint is enforced at all times, on every IRIS node—i.e., it is part of the ADT’s semantics. To achieve the required preclusion semantics, the constraint is checked as a precondition to every IRIS operation that can change the value of an application node’s operands. If the operation would violate the constraint, it is not allowed and a diagnostic exception is raised.

**Concurrency control:** In most cases, IRIS graphs are accessed as a unit—applications retrieve the root of the graph and traverse downward. This suggests that, for purposes of both concurrency control and clustering, all of the nodes that are navigationally reachable from any accessed IRIS graph node should be treated as a unit (e.g., all reachable nodes should be locked and transferred as a group to the application for subsequent manipulation). Pessimistic concurrency control mechanisms like multi-granularity locking [16] appear to be well-suited to ADTs like IRIS. These techniques implicitly lock all of the proper descendants of an object \( o \) when
is locked. This reduces the number of locks that are acquired, which reduces the overhead of locking.

IRIS graphs themselves are fine-grained objects, and an application may invoke multiple operations on any given node in the graph during a traversal. This suggests that the storage-level concurrency control architecture described in Section 6.3 is more appropriate than the ADT-level concurrency control architecture, since the storage-level architecture minimizes manipulation costs.

8.2.2 Associated Attributes

Different applications need to associate various kinds of information with nodes in IRIS graphs. Some examples of these kinds of information are:

- the name of the file containing the source code from which the IRIS graph was created
- a time stamp that indicates when the IRIS graph was created
- the line number in the source code at which a statement or identifier represented by a given IRIS node occurred
- any comments that appeared in the source code near the statement or identifier represented by an IRIS node
- annotations for purposes of parallelizing code [34]
- assertions or other information used for analysis [35]

From the perspective of applications that use them, these pieces of information are (conceptually) attributes of IRIS nodes. The information is not, however, part of the definition of IRIS. Further, not every node in an IRIS graph need have any particular kind of information associated with. Thus, we refer to these pieces of information
as associated attributes—application-specific information that is associated with, but not part of, IRIS nodes.

The ability to define associated attributes is a very important one throughout the language processing tool set, since many applications need to have application-specific information at their disposal to perform their tasks (or to reduce the time it takes to perform their tasks). For example, control flow graph nodes may be annotated with def/ref information and data flow information. It is important for applications to be able to annotate objects with application-specific information, without having to change the definition of the ADT to which the information is added.

**Type model:** Associated attributes are modeled using Pleiades relations and relationships. Each attribute is represented as a relationship between an object and one or more values for the associated attribute(s). These relationships are then collected into relations, which can be queried to determine whether a given object has an associated attribute.

**Navigational and associative access:** In general, associated attributes are accessed first by query, and then by navigation. The query occurs when an application asks whether or not a given object has an associated attribute. If it does, the relationship between the object and its associated information is returned to the client application, which then traverses the relationship navigationally to manipulate the associated attribute(s). Some client applications also iterate through entire collections of associated attributes—for example, a directory tool examines each associated attribute in a given collection.

The use of relations and relationships to model associated attributes is particularly useful because it allows application-specific information to be associated with an object without modifying the object itself.
Persistence: Associated attributes are typically gathered into collections of similar attributes. For example, the set of line number associated attributes for a particular IRIS graph might be collected together into a relation. Persistence almost always occurs at the level of these collections, on an all-or-nothing basis. Consequently, the default persistence model, navigational reachability, is appropriate.

The use of relationships to represent associated attributes is notable. Associated attributes are examples of pieces of information that are associated with other objects (e.g., IRIS nodes), but they should not be subject to the same reachability-based persistence as the “core” information that is properly part of the objects with which they are associated (i.e., the components of an object that must always be present for the object to have meaning; for example, IRIS application nodes must always have operands, but they need not have line numbers). Associated attributes become persistent independently of the objects with which they are associated. Relations and relationships proved to be very useful constructs with which to model associated information—they provide a natural means of delineating the extent of persistence, because they are not navigationally reachable from the objects with which they are associated.

Consistency management: Associated attributes are not subject to any special consistency constraints in the general case. For some specific kinds of associated attributes, however, uniqueness constraints are common—i.e., a given object can have only one associated attribute of a particular kind. An example of this is the source-file associated attribute. A node in a given IRIS graph represents a statement or literal found in a single source file; thus, it is not correct to permit multiple source file names to be associated with a single IRIS object. This uniqueness constraint is readily defined using Pleiades constraints with preclusion semantics.
We note that the need for preclusion semantics can be particularly important in the case of uniqueness constraints on associated attributes. This is because key indices are often kept on collections of associated attributes, to improve the performance of queries. Given the uniqueness constraint, it is preferable, for performance reasons, to use a key index data structure that presumes uniqueness of the key. If preclusion enforcement semantics cannot be supported, however (i.e., if a uniqueness constraint can be violated at any point), it is necessary to select a key index data structure that can allow duplicate keys, which imposes additional search-time overhead.

**Concurrency control:** In the experience of the language processing toolset developers, associated attributes tend to be written by a single application (usually the application that creates them) and read by most other applications. Two distinct access patterns exist, as described earlier: associative access to determine what (if any) associated attribute value a given object has, and iterative access through an entire collection of associated attributes. In the case of associative access, an ADT-based concurrency control architecture may be more appropriate, since it minimizes the amount of information that must be shipped back to an application. Particularly in cases where key index structures are maintained, an ADT-based architecture may significantly improve concurrency, since the key index structure must be locked while applications are using it. The use of a storage-level architecture would require shipping the index structure to applications, thus tying it up throughout the entire application’s access period. In contrast, using the ADT-based architecture means that the key index need only be locked while a query is in progress.

Two different models of iteration may be used—passive, where an application retrieves each member of a relation in turn and manipulates it, and active, where an application provides an action to be taken on each relation member (analogous to the mapcar function in LISP). The storage-level concurrency architecture is better suited
to passive iteration, since the application would otherwise have to take a one-IPC hit per relation member accessed; this cost can be reduced significantly by shipping the relation to the application.\(^2\) For active iteration, on the other hand, it is considerably less expensive, and may promote more concurrency, to ship the iterate action to an ADT server and allow the server to execute the action on each element of a relation.

### 8.2.3 SPECTRUM

As described earlier, IRIS is a language-independent abstract syntax graph type. While manipulating IRIS graphs in a language-independent manner is useful for some applications, tools that possess knowledge about the language in which a program is written may prefer to use that knowledge in viewing and manipulating IRIS graphs. For example, it is convenient for an application that performs Ada-specific abstract syntax graph manipulations to view the application node in Figure 8.1 as an [Assignment Node] with two operands named Left and Right. Using just the IRIS ADT definition, however, this tool could only determine that it is looking at an application node with two operands, and that the operator is a reference node whose value is “:=.”

SPECTRUM is a tool that produces language-specific views of IRIS graphs. It accepts descriptions of language-defined operators (such as assignment and if-statement) and their operands (e.g., “condition” and “else-part”), and it generates a collection of operations that permit applications to manipulate IRIS graphs using this language-specific information. Essentially, these operations map language-defined operators to application nodes with a given operator name and with appropriate numbers of operands, and they map operand names to positions within an application node’s operand list.

---

\(^2\)This assumes, of course, that most elements of a given relation are accessed during an iteration, which was found to be the case for associative attributes.
SPECTRUM does not itself produce a new ADT definition. The capabilities it provides must be modeled as views, because applications must have the ability to switch from language-specific to language-independent manipulation of the same IRIS graph.

PLEIADES does not really provide the ability to define views that are separate from an ADT definition. It provides the ability to define aliases and derived attributes, which are primitive capabilities that are useful in implementing views, but these were not really the right functionalities for implementing SPECTRUM. Instead, it would have been more useful to have the ability to have a specific view construct, which would allow application developers to describe a view (in the relational sense [30]) of a PLEIADES ADT, explicitly stating how the view maps to aspects of the ADT’s definition.

We note that the ability to define type-compatible views was fairly common among the PLEIADES applications we evaluated, particularly in the language processing tool set, where many of the program representations defined are language-independent data structures with language-specific views. The control flow graph ADT is another exemplar, where the structure of a control flow graph is language-independent, but the specific control constructs are language-specific.

8.2.4 Ada_To_IRIS

Ada_To_IRIS is a tool that translates Ada source code into IRIS. It is actually implemented as a two-phase translator: the first phase translates Ada to DIANA [38], an Ada-specific abstract syntax graph, while in the second phase, DIANA is translated to IRIS.

Translating DIANA into IRIS required the definition of relationships between DIANA subgraphs that represent declarative items and their IRIS counterparts. This was necessary to effect semantic resolution of uses of identifiers in IRIS graphs. The resulting relationships connect heterogeneous entities, since the DIANA ADT (like
the entire Ada_To_DIANA translator) was developed by a company that did not use Pleiades. The heterogeneity occurred both at the type definition level (the ADTs were developed using different object management systems) and the persistence level (DIANA uses a home-grown, file-based implementation, while IRIS uses Pleiades for persistence). Despite the heterogeneity, it was trivial to define this kind of relationship using Pleiades. One Ada_To_IRIS developer reported that the use of side-by-side name spaces in Pleiades (Section 5.1.2), and the accompanying persistence protocol (defined by the use of operations Get_PID and Get_OID), had greatly facilitated the use of a different storage management system for DIANA from that used in IRIS.

Only one significant problem arose in producing the DIANA to IRIS translator, and this problem is noteworthy. The DIANA specification does not mandate either value or object semantics for DIANA graphs. The implementation we obtained defined DIANA graphs to have completely value-based semantics. In fact, nodes in DIANA graphs had no properties at all that allowed us to identify nodes uniquely—the value of a given DIANA graph node changed each time it was retrieved from persistent storage. This meant that the relationships we defined between DIANA and IRIS declarators were inherently transient, because the DIANA nodes participating in these relationships became invalid upon the termination of program execution.

The fact that so many entities may have to be interconnected for long periods of time strongly suggests the need for identity of objects, a property imposed by the cross-cutting requirement for identity. The experiences of the Ada_To_IRIS developers, who tried to achieve interconnections among objects that could not be uniquely identified in a persistent manner, supports the inclusion of this cross-cutting requirement.
8.3 FLAVERS

As described in [39], “FLAVERS [35] is a static analysis tool that can automatically guarantee the absence, or detect the presence of, a wide range of user-specified properties or behaviors [in programs]. FLAVERS complements traditional testing approaches, which only demonstrate the presence or absence of errors for the specific test cases that have been executed. It also complements formal verification methods, which employ more comprehensive analysis, but require a great deal of time and expertise on the part of the user.”

Users describe the properties for which a program should be checked in terms of quantified regular expressions (QREs) QREs are deterministic finite automata, augmented with quantifiers that indicate whether the property should sometimes, always, or never hold in the program. The QREs are written in terms of sequences of events, where an event is a user-defined marker indicating the occurrence of something of interest to the user (e.g., a variable’s value is set, or a file is opened). FLAVERS instruments the nodes in control flow graph (CFG) representations of program components with the events that occur at those nodes. It then constructs a trace flow graph (TFG) from the instrumented CFGs. A TFG is a graph whose nodes represent the events that occur in the program, and whose edges represent flow of control between those events.\(^3\) TFGs may also have a second type of edge, called MIP (May Immediately Precede) edges. MIP edges represent event interleavings that may occur in a concurrent system. TFGs are later refined to collapse nodes that represent events not found in the QRE property specification. Finally, data flow analysis is used to check to see if the user-specified QRE is a string in the alphabet described by the TFG, and whether or not the specified quantifier holds.

Virtually all of the objects defined as part of FLAVERS are implemented using PLEIADES, either directly (by defining ADTs in PLEIADES) or indirectly (by using,\(^3\)Essentially, TFGs are forests of CFGs.)
for example, reusable components and other ADTs that were implemented using Pleiades). We describe the use of Pleiades in FLAVERS in this section.

**Type model:** To facilitate the analysis of programs, FLAVERS creates several representations of programs and information to be used to analyze the programs, as described above, and the FLAVERS developers found that these representations mapped readily to Pleiades type constructors. QREs are represented as Pleiades graphs. The association of events with CFG nodes is achieved using the associated attributes ADT (described earlier in this chapter). TFGs are represented naturally as graphs with two types of edges—one to represent flow of control, and one to represent MIPs.

For concurrent programs, FLAVERS annotates nodes in TFGs with collections of nodes, where each collection contains all of the nodes that could happen in parallel with that node (based on possible interleavings). This collection ADT, referred to as MHPs (May Happen in Parallel), was implemented using Pleiades relations that connect a TFG node, \( t \), with a set of other TFG nodes representing statements that can occur in parallel with \( t \). The set of TFG nodes is represented using a Pleiades relation.

To perform its analysis of TFGs, FLAVERS uses an iterative work list approach, where nodes to be evaluated are placed in a work list, from which they are eventually removed as they are analyzed. The work list is defined as a Pleiades sequence.

**Navigational and associative access:** FLAVERS makes fairly extensive use of both navigational and associative access over various objects. All of the graph structures are accessed navigationally, and iteration occurs over sequence and relation objects. Associative access occurs over several data structures. It is used to look up the IRIS node from which a given CFG node was created, and vice versa (using a relation that connects IRIS nodes to their corresponding CFG nodes). Given the
heavy use of associated attributes (e.g., to annotate CFG nodes with events, and to annotate TFG nodes with sets of MHP events), queries are also used to find annotations corresponding to particular objects. Finally, associative access was used to determine set membership. In all cases, the FLAVERS developers reported that only simple queries (i.e., selection based on the value of one field of a relationship) were needed for FLAVERS.

**Persistence:** FLAVERS predominantly uses a persistence-by-type model. The ADTs it creates are divided into those whose instances always become persistent and those whose instances never become persistent. QREs and TFGs always become persistent. In contrast, MHP sets never become persistent, since they represent an intermediate result used solely to help compute MIP edges. The Pleiades default navigational reachability-based extent of persistence was found to be the only appropriate model in all of the FLAVERS ADTs.

FLAVERS requires highly optimized object representations to perform its analyses more efficiently. One FLAVERS developer used the Pleiades persistence mechanism to define a specialized in-memory representation for some of the FLAVERS ADTs [35], in a manner similar to that used in the implementation of sets in the reusable components library (see Section 8.1). In particular, this developer used the Get_PID and Get_OID operations that provide the interface to the persistence mechanism to control the definition of separate in-memory and persistent representations. He provided his own implementations of these operations, and constructed the appropriately optimized in-memory representation upon invocation of Get_OID. The in-memory representations were marked as inherently transient, since they depended on execution-specific information, such as in-memory pointer values, and, at the end of a session, any modified persistent objects were copied back into their (un-optimized) persistent representation. This FLAVERS developer noted further that
the use of this (copy swizzle [75]) approach helped to minimize the performance overhead incurred, because the transformation happens once per tool invocation (twice for updated objects), but the objects are accessed in their optimized representations numerous times.

**Consistency management:** FLAVERS did not make use of the consistency management capability.

**Concurrency control:** Many of the FLAVERS ADTs are expected to be manipulated concurrently—for example, several developers may utilize the same reusable component in different software systems, so the various representations for that component (IRIS, CFG, TFG, etc.) would be shared if the developers ran FLAVERS concurrently on their software. Some of these ADTs may also be shared by other tools—for example, ProDAG (described in the next section) performs data and control flow analyses of programs, and it uses both the IRIS and CFG representations of programs as the basis of its analyses, and ProDAG might work in parallel with FLAVERS. In general, a fairly standard, single-writer, multiple-readers model of (competitive) concurrency control is expected to be appropriate for FLAVERS.

For several reasons, we expect that the storage-level concurrency architecture will be the only tenable approach to achieving concurrent access in FLAVERS. First, only persistent objects would be subject to concurrent access in FLAVERS (by definition, since all useful ADTs are subject to persistence-by-type, and intermediate results, which do not become persistent, are not likely to be accessed concurrently). Second, cooperative development is not expected to occur. Third, objects tend to be accessed in an all-or-nothing manner. For example, if FLAVERS retrieves the root of a TFG, it will usually traverse the entire TFG. Thus, it is relatively straightforward to define object clusters in a useful manner, which would significantly reduce the performance penalty associated with accessing objects concurrently (since, for example, the reach-
ability set of an object could be retrieved when the object is first accessed, with fairly high assurance that all reachable objects will be traversed, thus reducing the cost of traversing the object). Finally, performance is critical in FLAVERS, given the complexity of its analyses. This strongly suggests that it would be more desirable to accept the cost of reduced concurrent access (which occurs if large clusters of unused objects are retrieved) than to accept the performance penalties associated with finer-grained concurrency control. The storage-level architecture allows this tradeoff to be made easily.

8.4 ProDAG and TAOS

ProDAG (Program Dependence Analysis Graph System) [87] is a tool that analyzes program dependences. Program dependences represent the flow of information between statements or modules in a program. ProDAG analyzes three kinds of program dependences: data dependence, which refers to the dependence of a reference to a variable on a definition of that variable, control dependence, which refers to how one statement affects the execution of another, and syntactic dependence, which occurs when one part of a program is either data or control dependent on another part. Using program dependence information, ProDAG can statically detect various kinds of anomalies in a program—for example, the use of an undefined variable, or an unreachable path.

TAOS (Testing with Analysis and Oracle Support) [86] facilitates the process of testing software systems. It provides developers with support for defining and maintaining test artifacts (i.e., test cases, test suites (collections of test cases), test oracles, and test criteria), monitored test execution, formal behavior verification using test oracles, and coverage analysis (to determine which paths of a program a test suite exercises). TAOS is integrated with ProDAG—ProDAG determines which paths through a program must be exercised during testing to ensure that data, control, or
syntactic dependence coverage is achieved, and TAOS then monitors test execution to determine whether all required paths were exercised.

Both ProDAG and TAOS were built using Pleiades. ProDAG was built on top of the language processing toolset—it uses both the IRIS and CFG representations of programs as its starting point, and it extends the CFG with some additional information about data and control dependences. In particular, ProDAG annotates each node in a CFG with definition and reference (def/ref) information, which are collections of variables whose values are set or used in the statement represented by the CFG node. It then uses the def/ref information and the CFG to construct data, control, and syntactic dependency information as additional annotations on the CFG, which it uses as the basis for its analyses. For TAOS, all of the test artifacts noted above, and the interrelationships among them, were defined using Pleiades.

**Type model:** ProDAG predominantly used relationships and relations to model annotations on CFGs. Def/ref information was implemented using a relationship between a CFG node and collections of variable definitions and references that occur at that node. Dependences were represented as relationships between nodes in CFGs, and the collection of data and control dependences found in a particular CFG was gathered into a relation. Several different kinds of syntactic dependences (as defined in [83]) were also modeled as relations. These dependences were actually superset
tions of the data and/or control dependences, and they were modeled initially as Pleiades supersets (actually, superrelations).

TAOS used relationships and relations exclusively to model test artifacts, primarily because of a requirement for associative access (described later in this section). Test cases are defined as relationships with several fields, including the name of the test case, the name of a file containing the test data for the test case, the name of the file into which to write the results of executing the test case, the state of the test case
(e.g., passed, failed, untested), and various time stamps to indicate when the test case was last executed. TAOS supports two different types of test suites: \textit{manual}, in which a developer provides all test data, and \textit{random}, in which the test suite is populated by test data that is generated automatically from a description of legal inputs to the program to be tested. Both kinds of test suites are represented as relations, but their members include different information (e.g., random test suites specify the name of a file containing a context-free grammar that describes legal inputs to the program, while manual test suites include the name of the developer who specified the test cases).

While one developer reported that in general, the \textsc{Pleiades} type model provided the semantics required to define ProDAG and TAOS ADTs, he described a few noteworthy limitations. First, the model of subsets and supersets provided in \textsc{Pleiades} turned out to be too cumbersome for his needs. \textsc{Pleiades} originally defined subsets and supersets as a particular kind of predefined relationship between relations, with a predefined constraint enforced over that relationship (i.e., all members of the subrelation must be members of the superrelation). The decision to model subset/superset semantics in this manner was made because this model appeared to provide the most flexibility, dynamic control, and power. Unfortunately, the mechanism turned out to be too powerful and required too much application intervention; instead, the ProDAG developers would have preferred to specify, at object creation time, the subset/superset relationship. Second, a need for heterogeneous collections was noted. In particular, it would have been helpful in TAOS to be able to define suites of manual and random test cases. Heterogeneous relations are not currently supported in \textsc{Pleiades}, for reasons described in Section 4.1.1.3, though the need for them is recognized. Third, dynamic control over the definition of a type was found to be needed. For example, a tool that tracks bug reports (which is intended to be used with TAOS as part of the testing process) required the ability to add or remove fields
to individual bug reports. This capability is required as part of the object management framework (Chapter 3), in the form of the \texttt{Add\_Attribute} and \texttt{Remove\_Attribute} operations, but because Ada’s type model is static, these operations could not be defined readily in \textsc{Pleiades}. Fourth, the developer reported a need to determine the set of all instances of a type (i.e., operation \texttt{Extent\_Of} in the framework), for purposes of modifying instances of bug reports upon changing the definition of the report. For reasons described in Section 5.3.1, this operation was not supported in \textsc{Pleiades}. Finally, the developer noted that the repository abstraction (see Section 5.2.3) was not adequate for his purposes. Specifically, he wanted to be able to treat the repository like any other kind of collection ADT—he needed, for example, the ability to query a repository to determine whether or not it contained a particular object. He also noted that in the \textsc{Pleiades} applications he developed, he commonly wanted to define a directory object in each repository, where the directory would contain entries for all of the “root” objects from which applications might start traversing (e.g., a directory in a TAOS repository might hold references to all of the test suites in the repository). We note that other developers have made similar comments in the past (e.g., directories of all IRIS graphs contained in a particular repository are kept manually). This notion of having a “rooted” repository (i.e., denoting a particular object as the “root” or “directory” from which traversals over persistent objects begin) is also found in Mneme [74] and Napier88 [72].

\textbf{Navigational and associative access:} ProDAG and TAOS use both navigational and associative access mechanisms. ProDAG uses navigational access to traverse IRIS graphs and CFGs, to traverse the relationships it describes among CFG nodes, and to iterate over collections. It uses associative access to query the various collections it defines (e.g., to determine what variables are defined and used at a given node). TAOS also uses navigational access to traverse relationships and to iterate over collections,
and associative access to query test suites. One developer noted that ProDAG and TAOS usually wanted to find members of relations based on the value of one field.

The following comments were made with respect to the use of navigational and associative access in ProDAG and TAOS. First, ProDAG and TAOS commonly combined navigational and associative access—for example, the tools would first perform a query, then navigate through the set of returned members. Second, relations were found to be extremely useful constructs because they support n-directionality of references. Finally, it was noted that the ability to query graphs would have been a very useful capability.

**Persistence:** The majority of objects created by ProDAG and TAOS become persistent. ProDAG’s API provides instance-level control over persistence, though it was noted that the current version of ProDAG’s graphical user interface does not make this control visible to the human user. By default, TAOS makes all test artifacts persistent. TAOS users can, however, indicate circumstances under which the results of running test cases should or should not be made persistent (e.g., “save the results of all failed test cases”). This is a situation, like that in the case of the reusable components library, in which it would have been useful for developers to be able to indicate that objects should become persistent by default (instead of using Pleiades’ default, which indicates that objects are transient by default) and allow applications to determine when objects should not be persistent.

For both ProDAG and TAOS, the use of a navigational reachability definition for extent of persistence was generally appropriate. The only exception noted was in the case of some of the associated attributes. Essentially, ProDAG used an associated attributes model to achieve the effect of extending the definition of an object with additional attributes (since, as indicated earlier, Ada’s static type model did not readily permit the provision of operations to add attributes to objects). With
respect to persistence, however, associated attributes do not provide the right semantics. The semantics ProDAG required were for the associated attributes to become persistent if the object with which they are associated becomes persistent. By definition, however, associated attributes are not navigationally reachable from the objects with which they are associated, so they do not, by default, become persistent when their corresponding objects do. It was possible to work around this problem, but the developer would have liked either to have the AddAttribute operation, or to have better control over the definition of extent of persistence.

**Consistency management:** Consistency management was used in TAOS, but not in ProDAG. In TAOS, it was used to implement up-to-datedness constraints among objects. In general, the constraints react to changes in certain aspects of the state of a test case or test suite (e.g., a test case’s status changes to “passed” from “untested”), and their corresponding actions propagate the effects of these changes to other, related objects (e.g., summary information about the state of a test suite). Some of the TAOS constraints also are used to enforce the desired object persistence semantics. For example, upon a change in a test case’s status, information associated with the test case (such as the output generated from executing the test case) may become persistent or transient, based on user-supplied information. The persistence status of the modified test case is changed as appropriate upon update to the test case’s status. Pleiades’ consistency management model was found to provide the necessary functionality for TAOS.

**Concurrency control:** For both ProDAG and TAOS, fairly traditional kinds of lock-based concurrency control are needed. One developer noted that he expects to need locking at the level of individual test suites for TAOS, and at the level of CFGs (and all associated information) for ProDAG. In general, a standard read/write locking scheme is expected to be appropriate, augmented with notify locks (as defined,
for instance, in ObServer [95]) that would be used, for example, to allow the ProDAG graphical user interface to update its depiction of a CFG appropriately if the CFG changed. Cooperative development is not expected to occur frequently, if at all, except in a very limited sense—namely, that one developer might be allowed to look at a TAOS or ProDAG object while it was in the process of being updated, but, in general, multiple developers should not be allowed to write to the same objects.

“Intention to create” locks may also play an important role, particularly in ProDAG, and potentially in other tools, like FLAVERS, in which object creation time is high. As one developer noted, one user may run ProDAG on module $M$. Creating all of the dependence information, and producing analysis results, takes some time. While ProDAG is running on $M$, another user, who uses $M$ in a different system, may decide to run ProDAG on $M$. In this case, it would be highly advantageous for the second user to be able to find out that a ProDAG analysis of $M$ is ongoing, so that (s)he can wait for the resulting objects to become available, rather than duplicating the effort to run ProDAG again on $M$.

Both ProDAG and TAOS may need some kinds of meta-data information about transactions. In particular, because they are interactive tools, ProDAG and TAOS need to be able to find out when other developers acquire locks on objects that are in use. They must also be able to determine whether an object requested by a human user is already locked (with a conflicting lock mode), and to find out when the object becomes available, since it is typically the case that a human user will respond to the unavailability of one object by performing a different activity in the interim. For purposes of facilitating off-line coordination, it may also be useful for ProDAG and TAOS to be able to determine which user holds a lock on a given object.

It was noted that concurrent access to both persistent and non-persistent objects is expected to occur in ProDAG and TAOS. For example, two developers might be assigned to work together to track down a bug in a system. They might start by
running ProDAG on the system, to detect various kinds of anomalies. Since they will modify the system as soon as they find the bugs, there is no reason to make the objects ProDAG creates persistent—they will be invalidated as soon as the system is modified—though the developers retain their ability to make these objects persistent at some later point, if they so choose. Both developers must access the transient analysis data ProDAG provides, however, which requires concurrency control. This suggests that, at least for these types of objects, there may be a significant cost associated with the storage-level concurrency control architecture, since the storage-level architecture would impose the non-negligible cost of persistence on concurrently accessed transient objects. Given that neither ProDAG nor TAOS will support much in the way of cooperative development, and given that both tools access large amounts of data at a fairly fine level of granularity, it is also unlikely that the ADT-level architecture would provide adequate performance. The sharing of non-persistent objects is likely to require an intermediate solution, perhaps incorporating a transient object concurrency control manager. Essentially, this component would behave like the storage manager in the storage-level architecture, except that it would not manage a persistent store. Concurrent access to transient objects would occur via this component. Experimentation will be necessary to determine whether this type of hybrid architecture would perform better than, for example, requiring objects to become persistent as a precondition to being accessed concurrently.

8.5 A Process Program for Booch Object-Oriented Design

Booch Object-Oriented Design (BOOD) is an object-oriented software design methodology [18]. The BOOD process program defines a representation of the BOOD methodology in the form of a program. The purpose of this process program is to facilitate the design (using BOOD) of software systems. Toward this goal, the BOOD process program helps (human) users to create and manipulate the various artifacts
of the BOOD process—namely, *class diagrams*, which represent the specifications of types (classes) in a software system in terms of relationships among types (e.g., subtyping, generic instantiation, and uses) the methods each type provides; *object diagrams*, which describe how objects (i.e., instances of classes) interact, in terms of messages passed, in a particular scenario by tracing the execution of a scenario; and *state diagrams*, which are finite state machines describing the behavior of a given class in terms of the states it can enter, the state-to-state transitions it can make, and the events that cause those transitions.

The BOOD process program is written in the process programming language Julia [110]. Pleiades was used, however, to define the BOOD artifacts, as described below.

**Type model:** The BOOD artifacts were represented in Pleiades primarily using relations, relationships, and sequences. Class diagrams are represented as `<name, classes, relationships>` tuples, where `name` is the name of the diagram, `classes` is a collection of class specifications defined within the diagram, and `relationships` is a collection of relationships (such as “is-a” and “uses”) among the `classes`. Each class specification is represented as a relationship that includes a collection of operations defined on the class (specified in terms of operation name, parameter list, and return type), a collection of attributes, a collection of instances of the class, and some additional information (e.g., a textual description of the class). Object diagrams are defined as `<name, objects, links>` tuples, where `name` is the name of the diagram, `objects` is a collection of objects used in the scenario described by the diagram, and `links` is a collection of messages that pass between the objects in the diagram. State transition diagrams are defined as `<class, states, transitions>` tuples, where `class` refers to the class to which the diagram corresponds, `states` is a collection of states the

---

*The descriptions of the artifacts presented in this section are accurate, but, due to the number of type definitions included in the specification of the BOOD artifacts (specifically, the product schema comprises eighteen types of relation, nineteen types of relationship, and two types of sequence), some details have necessarily been omitted for brevity.*
class can enter, and transitions is a collection of transition descriptions (i.e., the original state, the new state, and the event that causes the transition). All collections in all Booch diagrams are represented using Pleiades relations, since the Booch artifacts must be accessed predominantly via queries (described later).

One BOOD designer noted that the semantics of the Pleiades type constructors closely matched those required for modeling the Booch artifacts. Relationships were used to aggregate the various parts of a particular kind of diagram (e.g., the name, attributes, operations, and relationships of a given class definition) and to model the relationships among diagram components (e.g., to represent relationships among classes, such as subtyping). Relations were useful for defining collections of design entities (e.g., the set of operations in a class).

Navigational and associative access: The BOOD process uses both navigational and associative access. Navigational access occurs when relationship fields are traversed (e.g., to determine the type definition of an object in an object diagram) and when iteration occurs over sequences and relations. Associative access is used ubiquitously throughout the BOOD process, to find information relevant to a particular artifact and to answer user queries. One designer also noted that queries and navigation are most often used in combination when manipulating the BOOD artifacts. For example, the BOOD process can, upon request, produce an HTML page that describes the information contained in a given class diagram. Users specify the name of the class diagram, which is used to query a directory of class diagrams to find the appropriate diagram. From the diagram definition, the process navigates to the set of classes defined in the diagram and their interrelationships. The process can iterate over the set of classes, and, for each class, present the class’s definition; as part of this process, it may query the set of class interrelationships to determine the set of types with which a given class is associated.
Although first-class status and identity of queries is not currently required in the definition of the BOOD artifacts, a BOOD designer noted that this capability would be very useful in the implementation of a planned query server for the BOOD artifacts, to which applications would send query or predicate objects to be evaluated.

**Persistence:** The BOOD artifacts are essentially persistent by type—all artifacts become persistent by default, and users can choose to destroy extraneous artifacts. One BOOD developer indicated that it would have been simpler to implement the persistence-by-type semantics if they had been able to change the default value of the persistence attribute. This is analogous to the feedback received from the developers of the reusable components library and ProDAG/TAOS.

**Consistency management:** The BOOD artifacts are subject to numerous (ten at present, with more than fifty additional ones anticipated) consistency constraints. These constraints are all well-formedness constraints on the artifact definitions. They include uniqueness constraints (e.g., to ensure that no two classes or objects in a given diagram have the same name), enforcement of composition relationships, constraints on subclass relationships (e.g., a class cannot be a subclass of itself, either directly or indirectly, and no class can be disconnected from the subclass hierarchy), enforcement of coverage relationships between a class and its state transition diagram (i.e., the operations in the class should be able to produce all of the states and transitions found in the state transition diagram), and connectedness constraints (e.g., a state transition diagram must be a connected structure, such that each non-exit state has at least one next-state, and each non-start state has at least one previous-state). Many of these constraints are extremely complex. At present, the only action taken upon violation of any enforced constraint is to print an appropriate error message, though in the future, other actions may be taken in response to a violation (e.g., redoing the step that caused the violation or effecting repair of the violation).
The enforcement constraints on BOOD artifacts must occur on an instance-by-instance basis, since different constraints are appropriate for artifacts that are at different stages of the design process. One BOOD designer noted that Pleiades’ instance-based control over enforcement and relaxation of constraints was appropriate and necessary to achieve the required semantics.

Pleiades’ constraint enforcement mechanism was found to be too fine-grained to be appropriate for enforcing constraints on the BOOD artifacts. In particular, the number and complexity of constraints on the BOOD artifacts makes the cost of checking constraints very high, so the BOOD process program cannot tolerate the performance cost of checking these constraints upon each data access where they might be violated. Consequently, the BOOD artifact constraints are left unenforced for much of the time, and they are checked manually by the BOOD process program at appropriate points during the process. This points to several needs. First, it would be useful for the BOOD process to be able to associate constraint enforcement and relaxation with blocks of operations. This is similar to the model used in database systems, where constraints are relaxed at the beginning of a transaction and checked at the end of the transaction; it is also the mechanism defined in APPL/A [109], in which a block of operations can be performed while specific constraints are explicitly relaxed or enforced. At the end of the block, the constraints resume their original status (enforced or relaxed). Second, a BOOD process designer noted that while it was helpful to be able to check constraints for satisfaction at any time (i.e., the Is_Satisfied operation described in Chapter 3), it would also have been convenient to have an operation that checks a constraint and invokes the appropriate action(s) if the constraint is violated, though these semantics can be achieved manually.

Finally, a BOOD developer noted that the BOOD development team had had some difficulty in specifying inter-object constraints in Pleiades. Specifically, because Ada does not make operations first-class entities, constraints and actions in
PLEIADES, which are modeled as operations, are not first-class entities (this restriction and its implications are discussed in Section 7.1). This limitation, combined with Ada’s (and consequently, PLEIADES’) static type model, means that only those constraints specified as part of an ADT’s (static) definition can apply to instances of that ADT. To work around this limitation, the BOOD developers 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 the BOOD process program (with all of the usual concomitant adverse effects on maintainability and understandability).

Concurrency control: Since software design activities invariably involve multiple people (designers and managers), concurrent access to the BOOD artifacts is expected to be the norm. The BOOD developers anticipate that both competitive and cooperative concurrency control will be required. Competitive mechanisms are required, for example, to preclude conflicting updates to a product by concurrent design processes, while cooperative mechanisms will be needed to facilitate collaboration by developers and/or managers on various design artifacts (such as creating candidate classes and objects for a design). Further, one BOOD designer noted that he expects humans to have to be involved in some aspects of concurrency control—for example, to help resolve conflicts and to determine how to handle deadlock situations.

Since the BOOD artifacts are persistent by type at present, concurrent access is generally expected to occur to persistent objects only. A BOOD designer noted, however, that a future version of the BOOD process program might support dynamic control over the persistence of objects (e.g., so that only those artifacts found to be consistent at the end of one phase of the design process would become persistent). In such cases, concurrent access to transient objects would be necessary.
The BOOD artifact ADTs were implemented using precisely the ADT-level concurrency control architecture—all Booch artifacts reside at the Booch product server, and clients interact with objects via IPC. In fact, the decision to use this architecture was made primarily for reasons other than concurrency control. Different parts of the BOOD process program are written in different languages (e.g., the BOOD artifacts are written using Pleiades; the main process program itself is written in the process programming language Julia [110]; and the user interface is written in Java); thus, to facilitate the integration of components written in different languages, the components interact across a client/server split. Since this architecture also facilitates cooperative development, we expect that the BOOD developers will gain a dual advantage from using this architecture.

8.6 Agenda Management System

The Agenda Management System (AMS) facilitates cooperation among, and communication between, the agents (i.e., developers, managers, tools, and processes) whose activities comprise the software engineering process. It accomplishes this task by supporting the creation and manipulation of agendas, which are collections of items that encapsulate information about tasks that each agent is to perform. Essentially, agendas are analogous to “to do” lists—they tell agents what they should do, and they may include information about the task, such as a deadline by which the task should be accomplished, relative priorities of tasks, tools that should be used to perform a task, etc.

Type model: The AMS defines several ADTs, all of which were implemented using Pleiades. It provides an agenda type, which contains a collection of agenda items and collections of attributes. Attributes are <name, type, value> tuples. Agenda items, which represent tasks to be accomplished, are collections of attributes. Each
attribute specifies information about the task. (De)composition is supported by allowing agendas and agenda items to be the types of attributes, which facilitates the modeling of subagendas and subitems.

Agenda item attributes are modeled using Pleiades relationships with name and value fields; the value field is represented as a variant, since attributes may have any of several different types. Agenda items are represented as relations over attributes, to facilitate associative access. Agendas are also represented using relationships—they contain a collection of agenda items (implemented using a relation) and a collection of attributes (since agendas may themselves have attributes).

The AMS developer reported that the Pleiades type model provided semantics that closely matched his needs. He noted two significant limitations, however: the lack of first-class status for types and operations, and the lack of dynamic control over type definition. Both restrictions are imposed by Ada, which does not make types or operations first-class entities and which defines a completely static type model. Taken together, these limitations meant that it was not possible to allow agenda item attributes to have any type of value, though it was extremely desirable to support this capability—some of the information he expected clients to save included operations (e.g., tools to use for a given task) and arbitrary user-defined types. Instead, attributes had to be specified with a predefined set of possible types. The developer noted that the meta-data operations Pleiades provides to determine the type of an object were helpful, since they allowed him to ascertain which of the predefined types a given attribute had. Ideally, however, he would have preferred to be able to allow each instantiation of the AMS to specify its own types of attributes, and to allow new types of attributes to be incorporated dynamically, as a project’s requirements change. Thus, both dynamic control and first-class status for all types of entities were required, and their absence caused significant limitations in the definition of the AMS.
Navigational and associative access: The AMS uses both navigational and associative access. Navigational access occurs primarily to examine the values of relationship fields and to iterate over relations. Associative access is used extensively, to select attributes with specified names, types, and/or values, and to retrieve particular agenda items, based on the name and/or type of one or more of their attributes.

Persistence: Persistence in the AMS is always controlled by (human or tool) users, who chooses when (and if) to make objects persistent. Thus, both instance-based control and dynamic control were important for the AMS.

Consistency management: The AMS did not make use of consistency management. It has been noted, however, that if projects could specify their own types of agenda items, they might also want to specify constraints on or among agenda items or attributes. Dynamic control would, therefore, be extremely important in this context.

Concurrency control: Objects defined using the AMS are expected to be subject to pervasive concurrent access. For example, the task “track down the cause of a reported bug” might be placed on several developers’ agendas. The first available developer might then start working on the item, causing the item to be removed from, or updated on, the other developers’ agendas.

While competitive concurrent access to AMS objects may occur at times, it is expected that cooperative access will predominate, since the AMS is intended primarily to facilitate cooperation among agents involved in a software engineering process. Thus, the AMS developers anticipate that (sub)agendas, agenda items, and even agenda item attributes will be shared among different clients, and that these clients will communicate, in part, by concurrently reading and writing shared AMS objects. Further, interactions with the AMS are not expected to fit any sort of transactional
model at all—agents may expect to have access to their agendas at all times, to see immediately all changes that affect them, and to work cooperatively on specifying or updating agenda items.

The AMS developer indicated that he expects to require the ability to achieve concurrent access to both transient and persistent objects. Concurrent access to transient objects might occur, for example, in cases where two developers create AMS objects intended for private communication, or where some developers create an object as a temporary “note” to themselves. In all cases, the transient objects would have the ability to become persistent at a later time; thus, for the AMS, concurrency and persistence are expected to be orthogonal.

Given the concurrency requirements of the AMS, we expect that some variant of the ADT-based concurrency control architecture will be necessary, as this architecture ensures both that the desired degree of cooperation can be achieved, and that persistence can be treated orthogonally to concurrency. We note, however, that because human users will view their agendas through a graphical user interface, many agenda object values will actually have to be buffered at the client. This is likely to mean that some kind of update notification will be required, to inform clients when they have to reread object values.

8.7 Chimera

Chimera [3] is a heterogeneous open hypermedia system. It facilitates the definition and management of hyperlinks among related artifacts in a software engineering environment. The Chimera developers had several goals in producing the Chimera system. First, they wanted to support pervasive heterogeneity. In the context of a hypermedia system, heterogeneity may occur in the form of object definition languages (i.e., defined in different programming languages or object management systems). It may also occur in viewer implementation languages, where viewers are the tools that
are used to depict, and possibly edit, objects; for example, Emacs is a viewer for text documents. Second, because they wanted to address the needs of users of software engineering environments, the Chimera developers imposed a requirement for a high degree of concurrent access, both to objects (i.e., any given object may be manipulated by multiple viewers, and multiple objects may be depicted by a single viewer) and to a hyperweb. Third, they wanted to support composition of hyperlinks, to help structure and abstract large and complex collections of interconnected information. Finally, they wanted to provide developers with the ability to define $n$-ary links (i.e., collections of anchors) among artifacts, instead of the standard binary linking capability that most hypermedia systems support.

To help achieve the heterogeneity and concurrent access goals, Chimera was implemented using a client/server architecture. The Chimera server provides services that allow clients to define and manipulate hyperwebs and manage hypertext events. It also manages information about each client—for example, which viewers are running, what objects they are depicting (also referred to as a view), what anchors (i.e., regions of interest in a view) they define, etc. Chimera clients include one or more viewers, and each client defines its own notions of “object,” “view,” and “anchor.” In accordance with the heterogeneity requirement, the Chimera server does not manage client objects—clients are free to define their objects using any object management system. The only requirement Chimera imposes on client objects is that they must have identifiers with which Chimera can refer to them. Clients are also responsible for maintaining a mapping from an anchor to a specific region of a view.

All objects in a Chimera hyperweb are defined using Pleiades. We analyze Chimera’s use of Pleiades below.

**Type model:** Chimera hyperwebs include several kinds of entities. Views are essentially $<$ viewer, object, instance_id $>$ tuples, where viewer is a tool that is
depicting object; since the same viewer may be depicting any given object multiple times, instance_id is an identifier that differentiates one running instance of a < viewer, object > pair from another. To facilitate associative access, views are defined as PLEIADES relationships, and all views for a particular hyperweb are collected into a relation. Anchors are specified as < view, id > pairs, where id is a unique identifier assigned to each anchor by Chimera (since a given view may contain multiple anchors). Again, to facilitate associative access to collections of anchors, anchors are defined as relationships in PLEIADES, and the anchors for a given hyperweb are collected into a relation. Links are modeled as collections of anchors. Objects, views, viewers, anchors, and links can all optionally have a collection of associated attributes, which can contain any information a client wants to associate with these objects. Attributes are modeled as relations over < name, value > pairs.

Particularly for Chimera, the ability to model these kinds of objects as first-class objects with identity was crucial. For example, satisfying the goal of providing composable hyperlinks means that it must be possible to define anchors on a given link; similarly, to associate attributes with an object, it must be possible to identify the object uniquely.

Although Chimera itself does not provide dynamic control over the type definitions of hyperwebs, one Chimera developer noted [2] that another hypermedia system (Hyperperform [125]) does provide the ability to modify types dynamically. Thus, the need for dynamic control over type definitions is present in hypermedia support systems.

Navigational and associative access: The Chimera server alternates associative and navigational access to hyperwebs. It uses queries to determine, for example, the set of anchors to which a given anchor is linked, the viewers for a given object, and the value of a named attribute associated with a particular object. Following a query,
navigational access might be used, for example, to examine the values of relationship fields and to iterate over collections.

Chimera performs primarily simple queries—i.e., queries based on the value of one field of a relation. We note that, to improve the performance of its queries, Chimera defined many relation fields as key indices. In most cases, Chimera was able to provide Pleiades with functions that map key values to integers. These were used in the construction of hash tables, which were used to speed up queries.

Persistence: All of the Chimera hyperweb objects are persistent by type, and navigational reachability proved to be the correct model for extent of persistence for these types. In addition, Chimera defines some auxiliary types for its internal use that are inherently transient; for example, it defines a relation that connects viewer names with executables.

Consistency management: Chimera enforces some well-formedness constraints on hyperwebs. In particular, the lifetimes of some types of Chimera objects are bounded by the lifetimes of others, and the existence of some types of objects affect the definitions of others:

- Anchors are defined in terms of a view, which means the lifetime of an anchor is bounded by the lifetime of its view—once the view is destroyed, the anchor is meaningless and should also be destroyed.
- Since views are specific to a viewer and one or more objects, the view’s lifetime cannot exceed that of the viewer or any of the objects.
- A link comprises multiple anchors, so upon destruction of an anchor, the anchor should be removed from all links in which it participates.
All of these constraints have “roll-forward” semantics—that is, they are expected to be violated during the normal course of hyperweb evolution, and, upon violation, an appropriate repair action can be taken to reestablish consistency.

The lifetime boundary constraints are particularly interesting, because they actually affect the persistence model Chimera uses. That is, the navigationally reachable definition of extent of persistence suggests that if an object, \( o_1 \), becomes persistent because it is reachable from another persistent object, \( o_2 \), then when \( o_2 \) ceases to be persistent (e.g., because it is destroyed), \( o_1 \) will also cease to be persistent (unless \( o_1 \) is navigationally reachable from another persistent object). Although navigational reachability is used in Chimera in determining which objects become persistent, it is not used as the sole definition of end-of-persistent-lifetime semantics; in fact, although \( o_2 \) may determine when \( o_1 \) becomes persistent, \( o_1 \)'s destruction may signal the end of \( o_2 \)'s life. For example, a view refers to an object and a viewer; thus, if the view becomes persistent, the object and viewer objects would also become persistent. If the object or viewer stop being persistent (i.e., they are destroyed), however, the view object becomes meaningless and should no longer be persistent. These semantics are the inverse of the existence constraints and composition semantics that many databases provide. For example, PCTE [41, 119, 63] provides several specialized kinds of link types. One of these forces both endpoints of a link to continue to exist for as long as the link between them exists; another indicates that if the source of the link is destroyed, the target will be destroyed. Further, these are not the semantics provided in memory management systems, in which, for example, the existence of a view that refers to an object and a viewer would preclude the destruction of either the object or the viewer.

Chimera’s well-formedness constraints suggest that the definition of “extent of persistence” must be further subdivided into “extent of becoming persistent” and “extent of becoming transient,” since the two need not have the same definition.
Similarly, for purposes of memory management, navigational reachability cannot be the sole determinant of which objects are still meaningful and which are not.

**Concurrency control:** One of the stated goals of Chimera is to facilitate highly cooperative, concurrent development of, and access to, hyperwebs—multiple developers may share both hyperlinks and artifacts. Concurrent access to artifacts is outside the scope of Chimera, because Chimera deliberately does not manage objects (to support heterogeneity, as described earlier), but concurrent, cooperative development of hyperwebs is partially supported in Chimera, with better support anticipated in the future. As discussed in [3], developers are expected to modify links simultaneously; this may happen while other developers are using the links, and the modifications will have to be propagated appropriately. For access to other kinds of Chimera objects, a standard competitive concurrency control model (concurrent-reads-exclusive-writes) is expected to be appropriate. As in the case of the Agenda Management System (see Section 8.6), interactions with Chimera hyperwebs are not expected to fit a transactional model—developers may add, destroy, and traverse links at any time while they are working, and they will want to have all relevant changes become visible to them immediately, and to be able to traverse or update a hyperweb at any time.

In its current form, Chimera implements precisely an ADT-level concurrency control architecture—the hyperweb ADTs reside at the Chimera server, and clients interact with server-resident objects using IPC. This architecture was found to be the best one because it facilitates cooperation. The Chimera developers noted in [3], however, that they have found that they require an even greater amount of concurrency, so they intend to move to the most extreme form of the ADT-level concurrency architecture: one server per ADT instance (in this case, one Chimera server per hyperweb). The experiences of the Chimera developers help to validate the utility of the ADT-level concurrency control architecture to facilitate cooperative development.
8.8 Chiron

Chiron [117, 118] is a user interface development system—it supports the development and execution of graphical user interfaces. Chiron was designed with several goals in mind. One of its primary objectives was to achieve a strong separation between application code and any user interfaces written for that code. The result of this separation is that user interfaces and applications can be developed and maintained independently, thus reducing the cost associated with the development and evolution of user interfaces. Facilitating the development of heterogeneous applications was another major goal of Chiron, since heterogeneity was expected in the application and user interface development languages, as well as in the selection of toolkits (e.g., Motif or OpenLook) and other low-level support systems. Finally, the Chiron developers wanted to support the development of highly concurrent user interfaces. Concurrency serves several purposes. First, it allows a single application to have multiple user interfaces that are simultaneously active, multiple applications that are running in parallel to have a single user interface, and concurrent execution of an application and its user interface(s). By allowing applications, user interfaces, and the Chiron run-time support system to run concurrently, the Chiron developers hoped to reduce the overhead associated with adding graphical user interfaces to applications. Second, given the goal of completely separating applications from user interfaces, the Chiron developers did not want to impose a particular application architecture (e.g., event loops) on Chiron clients, as most other user interface systems do. The concurrent execution of the Chiron run-time system, applications, and user interfaces facilitates this separation.

The Chiron system architecture attempts to satisfy all of these requirements. Chiron employs a client/server architecture. Chiron clients are applications that manipulate ADTs. Any object that is to be depicted as part of a graphical user interface must be modeled as an ADT, where all access to the object must occur
via operations defined on the ADT. Manipulation of depicted objects causes event notifications to be sent to a dispatcher, which, in turn, notifies one or more artists that depict object(s). The artists, in turn, update their depictions as appropriate, based on the event that occurred. If a human user interacts with a graphical user interface in such a way as to require manipulation of the underlying objects, the artist associated with the user interface calls appropriate ADT update operations (which may, in turn, result in further events notifications to dispatchers). The Chiron server essentially defines an abstraction that encapsulates all of the functionality needed for artists to produce graphical depictions, without exposing any of the specific details of the particular underlying graphical substrates (e.g., X windows or Motif). The Chiron server distinguishes two kinds of depictions: abstract depictions, which is what artists produce (e.g., an abstract depiction might include various kinds of shapes and lines between them), and concrete depictions, which is how the abstract depiction is actually drawn on a display device (e.g., in terms of pixels, colors, or other primitive drawing abstractions provided by the low-level graphical substrate). The Chiron server keeps the abstract and concrete depictions up to date with respect to each other.

Chiron was developed prior to the production of PLEIADES, so PLEIADES was not originally used in Chiron. In a later version of Chiron, part of the Chiron server—i.e., the one dealing with abstract depictions—was reimplemented using PLEIADES, but no other reengineering was performed. In the remainder of this section, we describe the use of PLEIADES in implementing part of Chiron’s support for abstract depictions. We also discuss some of the more interesting requirements a further reengineered Chiron would impose on object management support.

Type model: Chiron provides a language, ADL (Abstract Depiction Language), in which types are defined that can be used in creating abstract depictions, and a
hierarchy of abstract depiction types (ADH). The basic idea is that if clients have particular kinds of graphical objects they create often, they can define types in the ADH for those objects, and they can then create instances of these types. For example, if a developer creates an artist that presents graphical depictions of cubes, the developer could create a Cube abstract depiction type, which would encapsulate information about how to draw cubes.

The symbol table for the ADH, which contains information about each type in the ADH and its set of operations, is defined using Pleiades. The symbol table itself is defined as a relation over <name, unique_id, operations> tuples, where name is the name of an ADH type, unique_id is an identifier assigned to each type by the Chiron server, and operations is a relation containing the definitions (i.e., name, parameters, etc.) for each of the operations defined on the type. Among the operation definition information is the parameter list, represented as a Pleiades sequence. A separate relation is defined to maintain the parent/child inheritance relationships among types in the ADH.

Navigational and associative access: The symbol table and inheritance relation are accessed primarily via associative access. Navigational access is used to examine information about any type retrieved from the symbol table. Information about a type’s operations is accessed both associatively (to find the definition of a particular operation that an artist might want to invoke) and navigationally (by iterating through the collection of operations).

Persistence: The symbol table and inheritance relation are persistent by type—all instances of these types must become persistent. Navigational reachability was found to be the appropriate definition for extent of persistence.
Consistency management: Although Chiron does not make use of the PLEIADES consistency management mechanism, it maintains several consistency definitions manually. It implements a bidirectional up-to-datedness constraint between corresponding abstract and concrete depictions. It also manually maintains a semantic integrity constraint between the inheritance relation and the symbol table. That is, when a type is removed from the ADH, it is removed from the symbol table, which means that the inheritance relation must be updated accordingly to remove all parent/child relationships in which the removed type. Chiron artists also maintain up-to-datedness constraints between actual application objects and their corresponding abstract depictions. Finally, various kinds of constraints are maintained among objects in abstract depictions. For example, if two circles are joined by a line, it might be desirable to enforce a constraint that anchors the line to the circles such that if either circle moves, the line changes shape to move with it.

It is not difficult to see how PLEIADES could be used to support Chiron’s consistency management needs. The mappings between application objects and their abstract depictions, and between abstract depictions and concrete depictions, could be defined readily using PLEIADES relations.\(^5\) (In fact, they are currently implemented using a hand-crafted table ADT.) The up-to-datedness constraints can then be defined in terms of those mappings, such that upon update to one object, its corresponding object can be identified and updated accordingly. The semantic integrity constraint between the inheritance relation and symbol table is also straightforward to define; as a postcondition to removal of an entry from the symbol table, all entries in the inheritance relation that contain a reference to the removed type are removed.

Two issues would be significant in using PLEIADES to manage consistency in Chiron. First, the container problem may be an issue in being able to maintain the

\(^5\) These relationships can be defined without requiring either application objects or concrete depiction objects to be defined within PLEIADES, since PLEIADES satisfies the heterogeneity cross-cutting requirement.
two up-to-datedness constraints, since these constraints must be checked immediately upon potential violation. Further, as noted in Section 7.2, the container problem is significantly complicated by the presence of heterogeneity, and facilitating the development of heterogeneous systems is one of the stated goals of the Chiron work. At minimum, we expect that application objects might be defined in other languages or object management systems, which would affect the maintenance of the up-to-datedness constraint between application objects and abstract depictions. Second, particularly in the case of maintaining position relationships among abstract depiction objects, performance may be a significant issue. Object depictions must remain correct as the objects are moved interactively by human users—that is, objects must track accurately; users will not accept perceptible time lags between when they think they have moved an object and when it actually moves on the display. This means that manipulation of objects must happen at a very fine level of granularity—a single user “move object” event might actually involve a large number of very small movements, to enable accurate tracking to occur. If positional constraints were defined in Pleiades, each of the small movements would result in a constraint violation and repair. It is unknown whether a general-purpose consistency management mechanism, such as that provided in Pleiades, could have the performance characteristics required in maintaining this type of constraint. Further study is needed to determine both the feasibility of using a general-purpose mechanism, and to what extent it is possible to use the support for heterogeneity (imposed by the cross-cutting requirements) to define a special-purpose consistency management implementation that is fine-tuned for an application like Chiron.

8.9 Summary

The evaluations we performed have some direct implications for the hypotheses underlying this work—i.e., that the object management functionality resulting from
satisfying the cross-cutting requirements provides appropriate support for the construction and evolution of complex applications. In this section, we discuss these implications.

8.9.1 Type Model

Completeness: The necessity of type completeness was most obvious in the significant limitations that were encountered when this requirement was not satisfied. For example, the failure to allow relations of arbitrary types was problematic for several of the reusable components and for ProDAG, which needed to define heterogeneous collections. Conversely, those type constructors that satisfy the completeness requirement were used without difficulty, and they were used to define some extremely complex structures (e.g., in the BOOD artifacts, FLAVERS, ProDAG, and TAOS).

Dynamic Control: Many tools required the ability to control the definition of types dynamically. The need for dynamic control occurred most often in the need to add attributes to, or remove attributes from, objects. For example, dynamic control had to be simulated using collections of essentially typeless attributes in TAOS (to add and remove fields of bug reports); dynamic control over an object’s set of attributes is achieved by modeling attributes as collections in the AMS (to dynamically define new types of attributes for agendas and agenda items), Chimera and BOOD (to add and remove attributes of objects), and all applications that use the associated attributes abstraction (e.g., CFGs, IRIS, ProDAG, and FLAVERS). Ada’s static type model, which does not provide the necessary form of dynamic control, proved to be a major problem for many applications, as evidenced by the extra effort most of these developers expended to simulate that control.

First-Class Status: The need for first-class status of types becomes very clear upon examination of some of the limitations, which originate with the failure to satisfy
this requirement, that developers reported. For example, the need for attributes to be first-class entities was found pervasively (e.g., in the AMS, BOOD artifacts, associated attributes, and Chimera hyperweb objects). In many cases, the attributes were shared among objects. Similarly, need for first-class operations was found. For example, Chiron clients must specify call-back operations to be invoked upon the occurrence of certain events; agenda items in the AMS may specify an operation to execute in order to perform a task; and the need to enforce operation invocation order constraints in reusable components required first-class operations.

**Identity:** Every Pleiades application described in this section relies heavily on identity of objects. The use of identity occurs commonly to facilitate composition and decomposition (e.g., in Chimera and the AMS), and to support shared substructure (e.g., in IRIS and ProDAG graphs, TAOS test artifacts, and BOOD design objects).

**Meta-Data:** The operations Pleiades defines to provide meta-data about Pleiades-specific types were used in several applications (e.g., to determine whether a node in an IRIS graph has the type Application or Reference, and to determine the extent of a type in TAOS). The lack of meta-data about non-Pleiades types (i.e., about Ada types, most of which do not come with meta-information) proved to be a notable limitation. In fact, several clients had to work around this limitation manually, by creating their own meta-data about such entities as operation definitions (e.g., BOOD and Chiron), type definitions (e.g., the AMS), and type hierarchies (e.g., Chiron and BOOD).

**Generality and Heterogeneity:** Heterogeneity is clearly the rule among the applications we evaluated, in terms of the type semantics the applications required. For example, some applications used relations with multiset semantics (e.g., TAOS), while others enforced uniqueness constraints to achieve standard set semantics (e.g.,
some reusable components and BOOD); some applications defined graphs as connected structures (e.g., IRIS and CFG), while others defined them as collections of nodes and edges (e.g., state transition diagrams in BOOD and the graph structures ProDAG defines). No developers reported having any difficulties in either selecting or implementing the type semantics they required, which suggests that the object management framework operations (as implemented by Pleiades) satisfy the generality requirement.

8.9.2 Navigational and Associative Access

Completeness: It is interesting to note that, although most developers indicated that they could envision needing complex, computationally complete queries, the large majority of Pleiades applications did not actually require complex queries—typically, their associative access requirements were limited to retrieval of objects based on the value of just one field of the object.

Beyond the computational completeness of queries, the completeness requirement suggests that it must be possible to define and navigate over any path between objects, and that it must be possible to formulate queries over arbitrary collections of objects, and both of these capabilities were found to be very important. The need to define and traverse navigational paths among objects was pervasive in all the applications we evaluated. The need to define and query any collections of any kinds of objects also turned out to be important in many applications, and the current Pleiades restrictions with respect to associative access (i.e., that it can occur only over relations, which, in turn, are restricted to having relationships as members) were problematic. For example, the implementation of the symbol table reusable component required queries over sequences, and many applications (including several reusable components, IRIS, CFG, and ProDAG) required the ability to query graphs. It is clearly very important for complex applications to be able to specify arbitrary, queryable collections.
**First-Class Status:** We note that first-class status of queries did not turn out to be needed widely, though one application developer (for BOOD) indicated that he would take advantage of such a capability in the future. First-class status of navigational paths, however, was definitely important to several applications. It was particularly important in cases where graph edges had to be attributed (e.g., CFG and ProDAG).

**Identity:** Some kinds of navigational paths did require identity. For example, identity of edges was important in CFGs, IRIS, and ProDAG because applications had to be able to associate edge-specific annotations with edges dynamically. As in the case of the first-class status requirement, identity of queries was not necessary for many clients, but, as described earlier in this chapter, the capability was noted as being important for the implementation of the BOOD product query server.

**Meta-Data:** The primary use for meta-information about how an object can be accessed either navigationally or associatively appeared to be in the implementation of general-purpose algorithms and “meta tools.” An example of the former is found in one of the reusable components, which defines a generic graph depiction algorithm. This component operates on the structure of any given graph type, so it uses meta-information about nodes and edges for a particular graph type to determine how to traverse and depict instances of the type. “Meta tools” are essentially tools that operate on other tools. An example of a meta tool is TESS [59], which performs type evolution of Pleiades ADTs. Though not evaluated in this section, TESS is also a Pleiades client, and it uses meta-information about the structures of types to compare old and new type definitions, and it determines how to navigate from one object to another during the process of transforming old instances to conform to the new type definition by using Pleiades meta-data operations.
**Dynamic control:** While some developers noted that they could imagine using the ability to perform ad-hoc (dynamically defined) queries, none actually noted the lack of this kind of dynamic control in Pleiades as a limitation. The ability to add or remove navigational paths from objects dynamically, however, did turn out to be an extremely important capability for many applications—e.g., all applications that use the associated attributes reusable component (CFG, IRIS, ProDAG, FLAVERS) and that explicitly model types as having collections of attribute objects to effect dynamic addition/removal of attributes (e.g., IRIS, AMS, Chimera, Chiron, BOOD).

**Generality and heterogeneity:** Given the fairly limited query semantics we encountered, the generality and heterogeneity requirements for associative access did not appear to be crucial for most of the applications we evaluated. This is moderately surprising, though we note that the need to define many more types of queries than we encountered in evaluating these applications has been well documented (e.g., [15, 8, 70]). The generality and heterogeneity requirements were considerably more important for navigational access, and different semantics were found to be needed for defining and traversing navigational paths. For example, some applications treated these paths as first-class entities and manipulated them as objects, while others simply retrieved the endpoint(s) of navigational paths.

### 8.9.3 Persistence

**Completeness:** The ability to make instances of any type of object persistent was very important among the applications we evaluated. Although some types of objects had semantics that did not allow them to persist past the execution of the application that created them (because they were either meaningless or extraneous), the majority of ADTs these applications defined made some or all of their instances persistent.
First-Class Status: As described in Section 5.1.1, the persistence attribute is not fully a first-class attribute. Some limitations reported by Pleiades clients can be traced directly to this restriction. In particular, several developers noted that they would have liked to have been able to specify a default value for the persistence attribute explicitly, which is an ability they have for all other kinds of attributes. This ability was not present in Pleiades, however, because of the persistence attribute is not modeled as a fully first-class attribute—in particular, although its value can be modified, its default value cannot be set explicitly. This limitation complicated the implementation of some necessary kinds of persistence semantics, particularly persistence by type and persistent-by-default in some applications (e.g., the BOOD process, ProDAG, and some of the reusable components).

Identity: Identity of the persistence attribute is not supported in the current version of Pleiades, for reasons discussed in Section 5.1.1. As a result, applications must use either the native support for navigational reachability-based extent of persistence, or, if they require alternate persistence semantics, they must employ the type-level mechanisms provided for controlling the extent of persistence. In some cases (e.g., ProDAG), the ability to specify extent of persistence through the use of shared persistence attributes would have greatly facilitated the implementation of desired persistence models. For example, this capability would have allowed ProDAG to define associated attributes as being in the persistence extent of the objects with which they are associated, which were the required semantics. Identity of persistence attributes provides this ability, as described in Section 3.2.3.

We note that, in cases where applications use a simple definition for extent of persistence, the full power of identity for persistence attributes may not be required. In such cases, there may be performance advantages for modeling persistence attributes
as values, rather than giving them identity, both in terms of space (since identity requires enlarging the persistence attribute) and access time.

**Meta-Data:** None of the developers we interviewed expressed a need to determine whether or not a given object is persistent. We suspect that this is, in part, due to the orthogonality of persistence—when persistent and transient objects have no visible differences, it is less likely that an application would need to know whether or not a given object is persistent. We expect, however, that because of the interaction between persistence and consistency management (described in Section 7), some kinds of consistency conditions will have to check for the persistence of objects—in particular, when the persistence (or transience) of an object can affect the satisfaction of a given consistency condition. Our experience to date, though, has been that most developers are careful to ensure that their persistence models cannot violate their consistency conditions (e.g., by using persistence by type or navigational reachability semantics).

**Dynamic Control:** The ability to make objects persistent dynamically was used fairly commonly. It was used, for example, throughout the reusable components library, in IRIS, and in the AMS. In several cases, however, the persistence decision was made either statically (e.g., using persistence by type semantics) or at object creation time. In such situations, there may be significant performance advantages to allowing this information to be stated statically, since optimization techniques are available for static and create-time persistence decisions that are not available for when dynamic decisions must be made (for example, separate persistent and transient heaps can be defined, as in [8]). Thus, the presence of static control, in addition to the dynamic control, may be beneficial.
**Generality and Heterogeneity:** We found that almost every persistence model we had seen used elsewhere had also been used in one or more of the applications we evaluated. Thus, the need for support for heterogeneity is clearly pervasive. Except where known restrictions in Pleiades caused (predictable) difficulties (as discussed above), developers reported that they had been able to achieve the persistence semantics they required readily, in accordance with the generality requirement. Further, the interface to the persistence mechanism was found to facilitate the use of alternate storage managers (e.g., one implementation of a dynamically sized strings package used one of Ada’s predefined direct-access file packages, while others used the default Pleiades storage management facility) and to enable the transparent use of special-purpose, optimized in-memory representations for objects. This strongly suggests that both the generality and heterogeneity requirements were extremely important constraints on any persistence model.

8.9.4 Consistency Management

**Completeness:** Computational completeness of constraints and actions turned out to be very important, as some applications specified some fairly complex consistency conditions and actions (notably, the BOOD process and TAOS).

In performing our evaluation, we noted that some kinds of consistency definitions were used fairly often—for example, uniqueness and referential integrity constraints. Although it is reasonably straightforward to define such constraints in Pleiades, some developers confirmed that they would have found it very convenient to have had a library of predefined (and potentially extensible), commonly occurring constraints like these, and to have been able use that library whenever possible. In general, we believe that developers would prefer to use the simplest possible specification for a constraint, using a computationally complete formalism to achieve more complex or special-purpose semantics, rather than stating all constraints in a computationally complete formalism. Several possible ways of simplifying more “standard” kinds of
constraint specifications exist, including the definition of a reusable constraint library and the use of a simpler, computationally incomplete constraint specification formalism in combination with a computationally complete one. We defer the examination of the relative advantages and disadvantages of these approaches to future work.

First-Class Status and Identity: Neither constraints nor actions were modeled as first-class objects with identity in Pleiades, due to Ada limitations (see Chapter 7 for details). While this restriction did not cause difficulties for many Pleiades clients, it proved to be particularly problematic for the BOOD process, which requires the ability to specify interrelationships among constraints and among actions. This is precisely the functionality that having first-class status and identity of constraints and actions would provide.

Meta-Data: Currently, no Pleiades clients make use of the operations that provide meta-information about constraints and actions. In many cases, this turned out to be because a given consistency definition was enforced on all instances of a type, or on a known collection of instances, which tends to obviate the need to ask, for example, whether or not the constraint is enforced on any given object. In one case (the BOOD process), meta-data are expected to be used in the future, but application development is not yet to the point where meta-data have been used.

Dynamic Control: We found dynamic control over enforcement and relaxation of constraints to be very important in some applications. For example, for some of the reusable components, dynamic control was needed to allow different applications to choose whether or not to enforce constraints on their own instances of ADTs (e.g., some sets might enforce a uniqueness constraint, while others might not). In addition, the BOOD process program uses dynamic control pervasively, to effect the enforcement of consistency definitions at appropriate phases of the design process.
We note that static control over enforcement of constraints also was found to be important in some cases. For example, all IRIS graphs, without exception, must satisfy the constraint that nodes have at most one parent. When some or all instances of an ADT must satisfy a constraint, it is considerably more convenient to allow developers to make this assertion statically.

**Generality and Heterogeneity:** The need for heterogeneity in consistency management was pervasive. It turned out to be important both at the semantic level (i.e., the ability to define different models of consistency), and at the implementation level (e.g., to be able to insert special-purpose mechanisms to meet the needs of clients such as Chiron, whose performance requirements preclude the use of general-purpose mechanisms). Developers reported that they were able to define the semantics they needed, and we found that developers had implemented a fairly wide range of consistency definitions, (potential) violation responses, and enforcement semantics. Only one developer indicated that he had found it difficult to achieve the enforcement semantics he required—as noted earlier, due to the high cost of checking its constraints, the BOOD process program needed the ability to indicate points in the program at which a particular constraint should be checked, rather than having the constraints checked at any point at which they could be violated. Essentially, this means that the BOOD developers want to be able to associate the checking of constraints with parts of the programs that manipulate the BOOD artifacts, rather than with the operations defined on the artifacts. This functionality can be achieved using the operations defined on type `Constraint` in the object management framework, but it is somewhat awkward to do so, which indicates that the object management framework did not, as specified, fully satisfy the generality requirement. As noted earlier in this section, we believe the BOOD developers’ experience suggests, at minimum, that a
new operation must be defined on type `Constraint`, which, upon invocation, checks one or more constraints and invokes all appropriate actions if they are violated.

**Other Issues:** In evaluating client usage of Pleiades, we have noted that four general classes of constraints are pervasive. These are:

- **Up-to-datedness constraints:** Up-to-datedness constraints are used to effect percolation of changes among related objects. For example, when an AST changes, any CFGs that are derived from it must change accordingly to retain the mutual consistency of the CFGs with respect to the AST. Often, up-to-datedness constraints are used to maintain “is-derived-from” relationships among objects. Examples of these kinds of constraints are found ubiquitously in the the language processing and analysis toolsets (e.g., to maintain up-to-datedness among ASTs and CFGs, CFGs and TFGs, and ASTs and def/use information).

Up-to-datedness constraints tend to be subject to roll-forward consistency management semantics—that is, it is expected that they will be violated during the normal evolution of objects, and, upon violation, a repair action may be taken to attempt to reestablish consistency. Failure to reestablish consistency is typically regarded as a problem to be addressed by further changes (perhaps with input from human users), and *not* by undoing the original action that violated the constraint. For example, modification of a source module invalidates its object code. Recompiling, if successful, will reestablish the up-to-datedness constraint between the source module and object code. Compilation failure, however, should not cause the original changes to the module to be discarded—instead, the errors are reported to the developer, who makes additional changes to correct the error. Thus, many kinds of up-to-datedness constraints are characterized by both the roll-forward consistency management semantics and the
fact that they may remain violated for some period of time while additional corrections are made.

- **Constraints related to operation semantics:** In several cases, developers used the consistency management mechanism to perform one or more actions before and/or after the execution of a given operation, thus modifying (usually extending) the semantics of those operations. For example, TAOS ensures the existence of a directory object in each persistent store. At the beginning of each transaction, TAOS needs to load the directory object for use during the transaction. To ensure that this object is loaded correctly, TAOS associates a constraint with persistent stores, such that upon invocation of the `Begin_Transaction` operation, the directory object is retrieved.

- **“Well-formedness” constraints:** These are constraints that impose type and instance semantics that are not expressible using specification mechanisms present in standard type models. For example, a well-formedness constraint on a the binary tree reusable component indicates that no instance may have cycles or shared substructure.

We found two special kinds of well-formedness constraints that appear to occur with some frequency. One expresses what we will call “beginning of life” semantics. These are activities that must occur upon object creation—if they do not, the object is considered to be invalid. An example of beginning of life semantics is found in TAOS, which, as noted above, relies on the existence of a directory object in each persistent store. If this object does not exist, the repository is unusable. To ensure that the directory object exists, TAOS associates a beginning of life constraint with persistent stores that causes a directory to be added to a new persistent upon creation. The existence of the persistent store is predicated upon the existence of the directory—if creation of the directory
fails, the persistent store is invalid and may not continue to exist. The other specialized kind of well-formedness constraint we encountered expresses “end of life” semantics. These enforce, for example, finalization and referential integrity semantics. Failure to enforce the appropriate end of life semantics may result in more global corruption of objects.

In most cases, well-formedness constraints are expected to be inviolate (at least outside of blocks comprising composite updates during which consistency may be violated temporarily), as they partially define the set of permissible states objects can take. If a repair action is specified and fails to reestablish consistency, the original modification is considered invalid and may be undone.

- **Ordering dependencies among operations:** Many kinds of full or partial order relationships among invocable entities (i.e., subtypes of type `Operation` in the object management framework) exist. For example, no `Pop` operation may occur on a stack object until the first `Push` operation 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 control the enforcement of, such constraints.

  Among the kinds of ordering dependency constraints we encountered in Pleiades clients were operation ordering, constraint check and action invocation ordering (e.g., in the BOOD product definition).

  This suggests that a consistency management mechanism should facilitate the specification and enforcement of these kinds of consistency definitions, and it should be able to take advantage of knowledge of characteristics of each kind of consistency definition to help select appropriate enforcement mechanisms. Further work is needed to determine how best to leverage this information.
CHAPTER 9
CONCLUSIONS AND FUTURE WORK

Based on the evaluation of Pleiades we performed, some conclusions can be drawn. First, when clients reported encountering significant limitations in their use of Pleiades, most of the limitations they noted can be traced directly to a failure to satisfy some of the cross-cutting requirements. Second, clients used extensively much of the extra functionality and flexibility that results from satisfying the cross-cutting requirements. Taken together, these two results imply that the cross-cutting requirements represent important constraints, at least on those areas of object management functionality we examined. Third, while we have not yet performed extensive performance analysis, anecdotal evidence indicates that even the minimally optimized performance of Pleiades was reasonable. This suggests that it is feasible to satisfy the cross-cutting requirements and achieve acceptable performance. We believe that this is, at least in part, an artifact of satisfying the orthogonality requirement, which ensures that clients incur minimal overhead for object management capabilities they do not use.

We have found that the goal of orthogonality is not completely realizable in all cases, due to interactions among object management functionalities that satisfy the cross-cutting requirements. Some of these interactions occur between persistence and consistency management (i.e., because the persistence status of an object may affect the satisfaction of a consistency constraint), between consistency management and concurrency control, and between the identity cross-cutting requirement and consistency management (i.e., the container problem). Nonetheless, the importance of
satisfying the orthogonality requirement to the extent possible is clear from analyzing the strengths and limitations of Pleiades reported by application developers.

Satisfying the cross-cutting requirements proved to be extremely important for many kinds of applications. The cross-cutting requirements have enabled a broad spectrum of application semantics to be supported readily, and, in several cases, they have facilitated the definition of special-purpose optimizations for applications with higher performance requirements. We have found, however, that the cross-cutting requirement for dynamic control provides more power than some applications need, and it comes at a cost, in terms of either performance or analyzability. For example, allowing dynamic control over the enforcement of constraints means that static analysis cannot be used to determine whether or not an object’s constraints are mutually satisfiable (because the set of applicable constraints may change dynamically, and because some of the constraints that could apply to an object may be mutually unsatisfiable, but they may not be intended to be enforced at the same time). We have discovered that, while the dynamic control mechanisms we included in Pleiades were generally quite powerful, neither Pleiades nor the object management framework we defined provided adequate declarative control mechanisms. For applications that do not require dynamic control over some object management functionalities, this meant that the applications incurred the cost of dynamic control mechanisms they did not use. We consider this to be a significant problem, because one of the most important reasons we imposed the cross-cutting requirements was to help achieve orthogonality, with the goal of ensuring that applications did not have to pay for capabilities that they did not need.

One lesson that has come out of this work, therefore, is that, while dynamic control is an important cross-cutting requirement, declarative control should also have been a requirement. The use of declarative control mechanisms enables better optimization to be performed, and it allows more kinds of application errors to be identified.
Thus, in cases where application developers have information available, they should be able to state this information declaratively. We believe that declarative control will help application developers better choose a position on the flexibility vs. efficiency spectrum that best satisfies their needs.

This work has led us to conclude that system development of the future will require database programming languages that represent instantiations of the object management model we described. The approach represents the one that best eliminates the burden on developers. It is not subject to impedance mismatch, and it does not require developers to shift from one programming paradigm to another as they interact with “database” vs. “programming language” objects—it permits developers to apply any feature of their programming language of choice to any kind of object they need to define or manipulate. Further, it permits static and dynamic optimization technology to be applied to ADTs and the applications that use them. Many kinds of optimizations are infeasible if the “programming language” and “database” parts of the compiler and run-time system are stand-alone black boxes—for example, it is not possible to perform garbage collection across a database/programming language boundary.

Several interesting issues remain to be addressed as future work; these are summarized below.

**Issues in Consistency Management:** Facilitating the enforcement of any required consistency management semantics is greatly complicated by the container problem—it is difficult to support general-purpose consistency management over first-class objects because a change to any object may potentially affect the consistency of any object that refers to it. Changes may also affect the definition of the “contains” relationship, which further complicates consistency management. The set of referring objects is defined transitively, which makes this set difficult and expensive to compute, if, in fact, it can be determined at all—particularly in a concurrent or
distributed environment, the set of referring objects may not be computable. With the increasing development of Web-based and other distributed applications, and of heterogeneous applications, this problem is becoming very serious and can result in a spectrum of problems (e.g., the inability to keep links up to date, to keep related Web pages mutually consistent, and to integrate components in a software system).

While numerous approaches have been taken towards addressing the container problem, none have been found to be adequate to solving all manifestations of the container problem. This suggests that a general consistency management mechanism must incorporate multiple strategies for solving the container problem, analysis techniques to help guide the selection of appropriate strategies, and appropriate specification mechanisms, which would allow developers to state properties about constrained objects that are useful in choosing the right enforcement strategy (e.g., that violation is permissible, as long as it is detected prior to any attempted manipulations of objects). Static analysis techniques (notably, data flow analysis) may be useful, for example, in helping to determine the set of operations that can manipulate objects in such a way as to violate consistency definitions. With this information, it would be possible to determine which enforcement mechanisms could be used, and then guide developers in selecting an appropriate mechanism, given other application requirements.

As noted in Chapter 8, in evaluating client usage of Pleiades, we have noted that four general classes of constraints are pervasive: up-to-datedness constraints, constraints related to operation semantics, well-formedness constraints, and operation ordering dependency constraints. This observation suggests that a consistency management mechanism should facilitate the specification and enforcement of these kinds of consistency definitions, and it should be able to take advantage of knowledge of characteristics of each kind of consistency definition to help select appropriate
enforcement mechanisms. We plan to explore this topic to determine how best to leverage this information.

**Full Support for Concurrency Control:** At present, Pleiades supports only a very primitive notion of transaction, and the current implementation does not support concurrent access or recovery. We plan to address this limitation in several stages. First, we will complete some experimentation with the concurrency control architectures described in Chapter 6, with the goal of determining more precisely what client characteristics affect the selection of a concurrency control architecture. We will then produce a mapping of the concurrency control part of the object management framework into Ada, ensuring that appropriate mechanisms are in place to allow application developers to select a concurrency control architecture that best satisfies their requirements.

**Support for Distribution:** Providing support for distributed object management is particularly important to facilitate the development of World-Wide Web-based applications. Some of the problems involved in distribution are very similar to those in concurrency control. Given the cost of communication across a network, it is important to try to maximize the amount of processing that occurs with respect to the amount of communication to amortize the communication costs. Achieving acceptable performance in the presence of distribution therefore requires addressing many of the same issues as in implementing persistence (Chapter 5), such as object clustering, and in selecting a concurrency control architecture (Chapter 6), such as where to cache objects. Thus, adequate support for distribution must incorporate knowledge about an ADT’s or an application’s semantics. We will attempt to define a framework in which analysis techniques can be combined with developer input to guide the distribution of objects (including applications).
Support for Additional Object Management Functionalities: The object management framework defined in Chapter 3 does not yet span the full range of object management functionality that complex applications require (as described in Chapter 2.2). We plan to expand the object management framework to include support for other necessary functionalities, including access control, versioning, configuration management, and type evolution.

Incorporate advanced compile-time and dynamic optimization technology: Although Pleiades’ performance was reported to be reasonable, the system does not perform many optimizations, and we believe it could benefit greatly from the application of advanced optimization technologies. These include, for example, using analysis techniques to optimize access to persistent [44, 45] or concurrently accessed objects (e.g., prefetching required objects, clustering and buffering objects effectively) [88], and using standard static compiler optimization techniques (e.g., code hoisting, inlining) to improve the performance of object manipulation [44]. We would also like to explore the use of various kinds of dynamic optimization techniques. That is, since different applications manipulate objects differently, it is unreasonable to expect that an optimization used to improve the performance of one application will be appropriate for all applications. Thus, we believe that applications and ADTs will have to be monitored to evaluate their access patterns and performance, and, as appropriate, dynamically change the implementation strategies underlying the ADTs to improve performance. Dynamic analysis and adaptation should be useful in a number of areas, including persistence (e.g., to redefine object clusters), concurrency control (e.g., to move from one concurrency control architecture to another), consistency management (e.g., to change the enforcement mechanism for one or more constraints), and navigational and associative access (e.g., to produce lookup structures that best optimize a particular kind of application access pattern).
BIBLIOGRAPHY


[59] Lerner, B. S. Type Evolution Support for Complex Type Changes. Technical Report TR-94-71, University of Massachusetts, Computer Science Department, Amherst, MA, October 1994.


[69] McCall, E. Personal communication, July 1996.


[76] Moss, J. E. B. Personal communication, July 1996.


[100] Stocks, P. Personal communication, August 1996.


