

PLEIADES: An Object Management System for Software Engineering Environments*

Peri Tarr
Lori A. Clarke

Software Development Laboratory
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract

Software engineering environments impose challenging requirements on the design and implementation of an object management system. Existing object management systems have been limited in both the kinds of functionality they have provided and in the models of support they define. This paper describes a system, called PLEIADES, which provides many of the object management capabilities required to support software engineering environments.

1 Introduction

Software engineering environments support the process of producing and maintaining software systems. One of the most common and pervasive activities of software developers is the creation and manipulation of software *objects* that represent artifacts of the software development process, such as requirements, specifications, designs, source code, test data, and analysis results. Objects that are created during the software development process tend to be large, complex structures with complex interrelationships to other objects. For example, design elements are related to the requirements they satisfy, source code is related to the design it implements, test data are related to the source code they test, and analysis results are related to the requirements, specifications, designs, or source code that were

analyzed. These objects must be maintained in a consistent state for arbitrarily long periods of time, perhaps while being shared by different tools and being accessed by multiple users.

A software engineering environment should therefore provide *object management* capabilities that facilitate the definition, manipulation, and maintenance of complex objects and their interrelationships. Because of the complexity and diversity of software artifacts and the processes that produce them, software engineering applications impose some challenging requirements on the design of an object management system. Programming languages, file systems, and database systems currently fail to satisfy these requirements. Recent work on database programming languages (e.g., [52, 1, 21, 13, 16]) and object-oriented database systems (e.g., [3, 7, 26, 60]) are attempting to overcome some of their limitations, but to date, none of these efforts have sufficiently provided the capabilities needed to support the spectrum of software engineering activities. Therefore, as part of the Arcadia project [56, 24], we have been trying to address these weaknesses to aid our own environment-building efforts, especially with regard to support for process programming [50] and software analysis [17, 40, 63].

This paper describes a prototype system, called PLEIADES (Programming Language Extensions Integrated with Advanced Database Extended Semantics), which provides many of the object management capabilities required to support software engineering environments. PLEIADES is a database programming language in that it extends a programming language (in this case, Ada [57]) with capabilities associated with traditional databases. It does not provide these capabilities in the traditional database style, however. Whereas database systems have commonly focused on efficiency and a strict model of consistency, rejecting capabilities that

*This work was sponsored by the Advanced Research Projects Agency under Grant # MDA972-91-J-1009.

violate these requirements, the PLEIADES system emphasizes functionality, flexibility, and ease of use. The result is an interesting and powerful system.

The remainder of this paper is organized as follows. Section 2 provides a small but typical example of a software engineering application to illustrate some of the object management needs that such applications have. This example is used repeatedly throughout the remainder of the paper to justify our requirements and to motivate our design decisions. Section 3 describes what we believe are important requirements on object management systems imposed by software engineering applications. Section 4 provides a brief overview of related work. A more detailed description of related work is given in Section 5, where each of the major language features of PLEIADES is described, justified, and contrasted with other approaches. Finally, Section 6 briefly describes our experiences using PLEIADES and discusses our plans for future work.

2 Object Management in Software Engineering: An Example

To illustrate some typical object management needs of software engineering applications, we describe here a subset of the capabilities provided by some Arcadia tools [11, 53, 40]. These tools create and maintain four major data structures: abstract syntax trees (AST), control flow graphs (CFG), definition and reference (def/ref) annotations, and dependency information [39]. 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 information is associated with CFG nodes. The 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 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 1.

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

¹ Only informal definitions are needed here. The interested reader should refer to [39].

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.

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 [38] 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 [31].

3 Object Management Requirements

The above example can be used to illustrate some of the functionality that should be provided in an object management system for software engineering environments, including high-level primitive type constructors, navigational and associative access over the same structure, persistence, consistency management, access control, concurrency control, resiliency, version control, configuration management, name management, evolution, and distribution. In addition, it demonstrates some "cross-cutting" requirements, which impose additional constraints on the ways in which the functional requirements should be satisfied.

The current implementation of PLEIADES addresses the first four functional requirements listed above and the cross-cutting requirements. Although the other functional requirements are not yet implemented, we have attempted to anticipate and plan for the incremental inclusion of these additional features. Both the functional and cross-cutting requirements are discussed below.

3.1 Functional Requirements

High-level type models: Tools that populate software engineering environments define and manipulate large amounts of complex data. Certain classes of abstract data types recur throughout software engineering environments: graphs, varying-length sequences, relations, and relationships. Graph objects occur, for example, in the form of abstract syntax graphs, control flow graphs, and dependence information graphs, as seen in

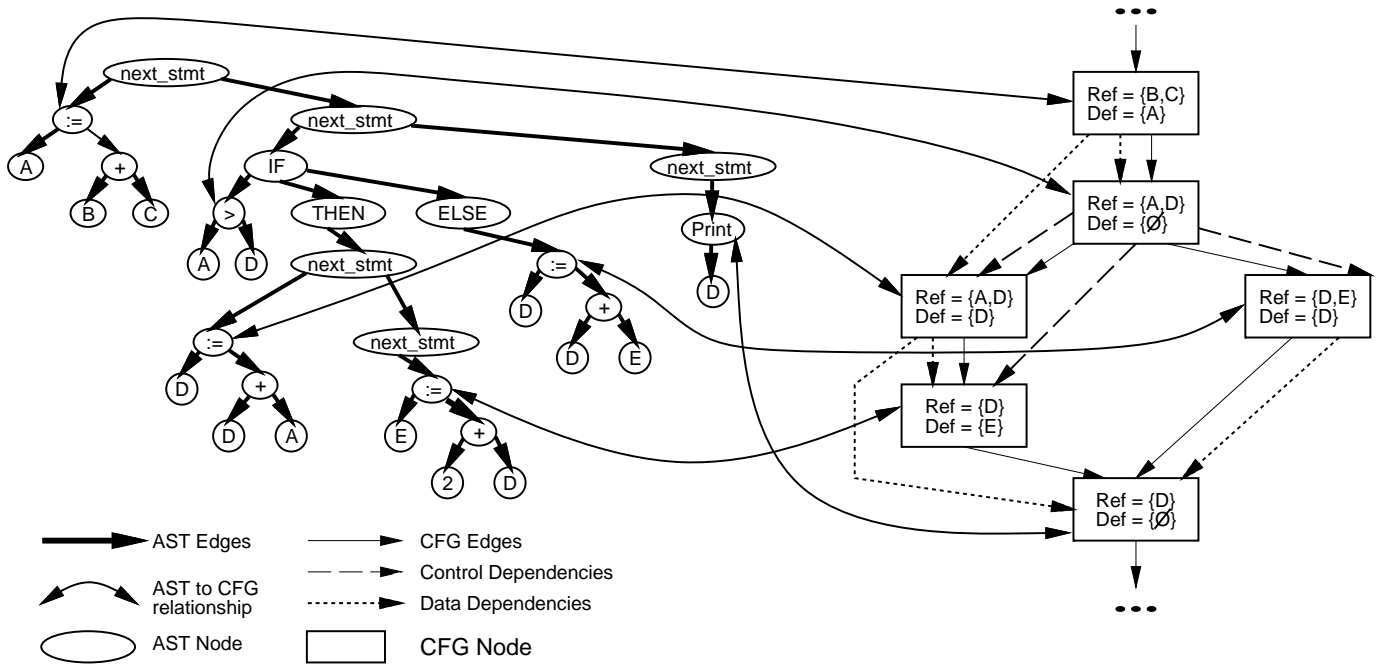


Figure 1: Example of AST, CFG, Def/Ref Annotations, and Dependency Information.

Section 2. The representation of ordered lists of objects, such as the sections of a document or the 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 may be connected to 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 a natural type for representing these interconnections, and relations are useful for gathering together related collections of relationships. In the above example, a collection of all the def/ref annotations can be used to answer cross-reference type queries.

Persistence: Some of the objects created during software development may have to persist for arbitrary periods of time (e.g., requirements specifications), while others may be transient (e.g., an intermediate result of a computation). The determination of which objects should or should not persist must rest with applications, since ultimately, only an application has enough information to make this decision. A persistence model suitable for use in a software engineering environment must therefore be flexible enough to support application control over the persistence of arbitrary objects [61]. 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 deleted. 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, could be deleted.

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 development environments tend to be traversed only navigationally or only associatively, while others, like a CFG, may have to be traversed both ways.

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. For example, a CFG for a module is consistent with respect to the AST for that module if the source code from which the AST was created has not changed since the CFG was created. Consistency management comprises definition of consistency conditions, identification of consistency violations, and reestablish-

ment of consistency following violations. Thus, consistency management is also a mechanism for supporting reactive control, whereby an action is invoked when some state or event occurs.

In the context of software engineering applications, a consistency management mechanism must be flexible enough to support complex consistency definitions, and to permit a range of consistency violation-detection and reestablishment activities. For example, given the above consistency definition for CFGs, one possible consistency management mechanism might identify the consistency violation *after* it had already occurred, and then reestablish consistency by invoking the CFG builder to recreate the CFG. For other kinds of objects, the violation of a consistency definition may be considered erroneous and must first be detected and then corrected, either by undoing the changes or performing some other corrective action, before proceeding. Further, different consistency definitions, violation identification, and reestablishment mechanisms may be applicable to a given object during different stages of development. 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. Thus, a consistency management mechanism for software engineering environments must facilitate the description of a wide variety of consistency definition, violation detection, and reestablishment mechanisms.

3.2 Cross-Cutting Requirements

The functionality provided by PLEIADES can be implemented using current programming language and/or current database capabilities, but not without extensive programming. For instance, the higher-level types can be implemented in most programming languages using a variety of low-level type constructors (e.g., with records, arrays, and pointers). However, a significant amount of code is required to implement these classes of types using such low-level constructs. Further, the development of these types is complicated and time-consuming, and it can be error-prone. Similar arguments can be made about persistence, navigational and associative access, and consistency management. Thus, an overriding concern in the design of PLEIADES is to provide a system that greatly facilitates the job of software engineering environment *builders*. Ease of use and flexibility are therefore important criteria that impacted our design decisions in attempting to satisfy the functional requirements. Ease of use and flexibility are vague criteria, however, that we have translated into more concrete requirements. These cross-cutting requirements, listed below, impacted all aspects of the design of PLEIADES and help distinguish our approach from other similar efforts.

Completeness: Computational completeness supports arbitrarily complex computations, while type completeness 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.

First-class status: First-class status provides the ability to treat all objects uniformly. The flexibility to pass a type or procedure as a parameter is an example of this requirement.

Identity: Object identity means that a given object can be referred to by multiple objects. Identity provides the ability for a change to a shared object to become immediately visible to any objects that refer to it. (Problems associated with the satisfaction of this requirement are discussed in Section 5.4.)

Dynamic control: The software engineering process is laden with decision points that must occur dynamically during the development or maintenance of an application. Thus, 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 important.

Meta-data: To make decisions dynamically, software engineering applications require information about their run-time state or environment. 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, and concurrency control models are more appropriate for supporting different kinds of applications and different phases of software development (e.g., [54, 24, 9, 51]). Our experience has led us to conclude that no single, high-level model will adequately support the diverse needs of software development. Within each of the functional areas of object management, different models are more appropriate for different projects and at different stages of development.

It is therefore crucial for an object management system for software engineering environments to be both *general*, so that it can facilitate the implementation of multiple models of support for any provided functionality, and *heterogeneous*, so that it can enable alternative models and implementation strategies to be used together in an integrated manner. In combination, support for generality and heterogeneity in an object management system allows software engineers 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.

4 Related Work

Object management needs have traditionally been satisfied through the use of capabilities provided by programming languages, file systems, and database systems. As we will describe below, existing systems have failed to provide many of the functionalities required to support software engineering activities and to address the cross-cutting requirements.

Early efforts to support the object management needs of software engineering applications can be categorized as either database or programming language approaches. The database approaches focused on using relational database systems, either alone or embedded in a programming language, for object management (e.g., [34, 45, 43, 23]), while the programming language-based work attempted to use file systems or low-level storage management systems to provide persistence of programming language structures (e.g., [6, 2, 61, 15]). Database systems support persistence and coarse-grained concurrency control and consistency management, but they fail to satisfy the other functionality requirements and except for provision of meta data, they do not address the cross-cutting issues. In addition, even in those areas of functionality they provide, their models of support are very restrictive (e.g., persistence and associative access apply only to relation or set types, and instances of these types are always persistent). The programming language-based approaches, on the other hand, provide persistence, high-level type models (though they provide limited support for high-level types of the kind used so often in software engineering applications), completeness, and identity, but they otherwise fail to address the need for broad-spectrum, flexible, integrated object management support.

Later research attempted to address some of these limitations. These efforts can be divided into database programming language and object-oriented database approaches. Database programming languages have generally attempted to extend programming languages with some of the kinds of functionality that are typically found in databases, such as set constructors, concurrency control, associative access, and persistence (e.g., [52, 18, 1, 21, 13, 16]). While database programming languages provide the computational completeness of traditional programming languages, they have tended to limit their support for persistence, associative access, and concurrency control to a subset of types (usually sets) and have provided highly restricted models of support. In addition, they frequently restrict the kinds of computations that can occur over database objects (e.g., only those actions that can be specified with a non-complete query language are permitted). Object-oriented database research, on the other hand, has attempted to introduce notions of inheritance, subtyping, and object identity to database systems (e.g.,

[3, 36, 37, 7, 26, 60, 58]). These systems are showing promise, but they have a number of shortcomings. First, they tend to be computationally restricted to enable query optimization; those that do not have this restriction usually overcome it by embedding the object-oriented database in a host language, which causes problems of impedance mismatch. Many of these systems do not support consistency control. Finally, these systems tend to be limited in their support for dynamism and in the generality of the functionality they provide.

5 Language Features to Support Object Management

The PLEIADES system has been developed to support our own efforts to build 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 3.1 and are of universal 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.

5.1 Appropriate High-Level Primitive Types

As noted in Section 3.1, 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.

5.1.1 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* 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.² 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.

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.

5.1.2 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

²While 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.

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.

5.1.3 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.³

Relations are unordered collections of relationships⁴ 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, for instance, some kinds of program dependence information are actually subsets of other kinds [39]. 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

³Note that relationships, which represent arbitrary “is-associated-with” connections, are semantically different from edges, which specifically represent “is-part-of” relationships. Thus, for example, while edges can be traversed implicitly or explicitly, relationships can be traversed only explicitly. Other semantic differences are described later.

⁴Relational database terminology defines relations as *tables* [19]. We use the more generic term *collection* to describe PLEIADES relations because of the semantic differences between the two relation models.

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. Meta data 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 they need not be unique. For indices whose types are strings or discrete types, indices can be built without programmer intervention. For other types of indices, programmers 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 2 shows some PLEIADES type definitions for the data structures depicted in Figure 1. 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`).

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 understand and maintain. The Gras [35] and IDL [46] 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 inte-

ger 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 [57], C [25], C++ [49]) 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., [3, 60]), but these are generally limited in the same ways that programming language arrays are (two notable exceptions are Gemstone [33], which provides an indexed set class, and EXTRA [58], 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 [10]. 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 leads 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. Finally, though the definition of subset semantics is possible in relational database systems, it requires varying amounts of programmer intervention.

Programming languages do not typically support either relation or relationship constructors directly. This shortcoming, in part, led to the development of database programming languages [5], 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 [50, 43, 45]. 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., [33, 7, 26, 58, 60]). Finally, we note that recent work on data structure precompilers to support software reuse [44] has suggested that the lack of a collection constructor in programming lan-

```

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 ( If_Stmt, For_Stmt, ... );

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 2: Partial PLEIADES Type Definitions for Example in Figure 1.

guages 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. 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.

5.2 Navigational and Associative Access

As discussed in Section 3.1, software engineering applications may want to traverse data structures navigational 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 3a shows a PLEIADES code fragment that declares instances of the **CFG_If_Node** and **If_Node_Edge** types, shown in Figure 2, and then performs a simple navigational traversal of the resulting CFG if-node.

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 3b 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 navigational 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 [59], 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

```

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 3: PLEIADES Code for Navigational and Associative Access to Instances of Types Defined in Figure 2.

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 [50, 18, 7, 60]. Some systems, such as [33, 58], 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. In addition, it 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. 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.

5.3 Persistence

PLEIADES defines persistence to be a property of instances, and this property is *orthogonal* to other properties of the instance [5]. Orthogonality means that the interfaces to persistent objects are identical to those of non-persistent objects [61], 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 [61]. Similarly, when a persistent object is retrieved from persistent storage, all of the objects reachable from it become available with no additional application intervention.⁵ Figure 4 demonstrates the use of the PLEIADES persistence mechanism. The root of the abstract syntax graph, `AST_Root`, becomes 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

⁵PLEIADES *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.

```

-- Make the root of an AST persistent:
AST_Root, AST_Retrieved_Root : AST_Node;
AST_Root_Persistent_Identifier : PID;
...
Get_PID ( AST_Root, AST_Root_Persistent_Identifier );

-- Retrieve the root of the persistent AST later:
Get_NPR ( AST_Root_Persistent_Identifier, 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: Using the PLEIADES Persistence Interface.

reachability definition based on the “is part of” 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 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” [55]. 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. For example, although the code shown in Figure 4 will cause the entire abstract syntax graph of Figure 1 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 `AST_To_CFG_Connections` (shown in Figure 3) 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 persis-

tent; e.g., [41]) and where all instances become persistent (e.g., [13]) 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., [43, 50, 34, 41]) 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, 21, 22], which provide dynamic control over the persistence decision, but not over extent of persistence, and the Ergo system [28], 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., [3, 58, 33]), tie persistence to types (e.g., [7]), or limit the types of objects that can be designated as “top-level” persistent objects (e.g., [30]); one notable exception is [60], which supports dynamic persistence decision over any type of object, but again, this system does not support control over extent of persistence.

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 `Get_PID` and `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 sub-component 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.

5.4 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.⁶ Constraints are computationally complete,

⁶In relational database terminology, this is called a *predicate*.

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,⁷ and an optional *action* that is to be taken upon detection of the violation (by default, an exception is raised).⁸ 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 [50].

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 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, metadata 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 (e.g., [57, 29]). In addition, programming languages typically support only

⁷Note 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.

⁸PLEIADES should not be confused with *constraint-based programming languages* (e.g., [12, 47]), which support problem-solving through the definition and satisfaction of sets of constraints.

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., [42, 32]) and exception handling mechanisms (as in Ada [57] and CLU [29]) 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 detect 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 enforcement⁹ or first class status or identity of the conditions or actions associated with these mechanisms.

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., [58, 4]), support limited consistency definitions, such as referential integrity (e.g., [33]) or programming-language-style consistency definitions (e.g., [3]), or support consistency definitions over only a subset of types (typically collection types; e.g., [50, 18]). A few active database systems, such as [48, 14, 30, 8], have included better support for constraint definition, but these systems have had a variety of limitations (e.g., [14] 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 [20], which defines rules to be first-class objects.

PleiaDES Limitations and Future Directions: The current implementation of PLEIADES does not sat-

isfy 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 and completeness. 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. Finally, the requirement for completeness has not yet been satisfied, since consistency definitions can be enforced only over graph, sequence, relation, and relationship types.

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.

6 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 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 3, there are some areas of functionality that we have yet to address. We have done preliminary investigation into most of these areas (e.g., [54]), and, in some cases, we have built independent prototypes [62, 27]. Although there is considerable interaction among the current and proposed capabilities, we feel relatively confident that our current system design

⁹PL/1 ON conditions do support dynamic enforcement.

will readily support these extensions.

PLEIADES has been used in about half a dozen projects within the Arcadia consortium (e.g., [40]), as well as by some industrial users (e.g., [31]). For the most part, users of the system have been very pleased with the functionality of the system and its overall performance.

Acknowledgments: This work has benefited from the ideas and contributions of a number of people. Jack Wileiden and Alex Wolf were involved in the development of earlier versions of the type and persistence models. Stan Sutton has provided useful comments on much of the work presented here, and we have benefited from his ideas and research. Debra Richardson, Margaret Thompson, Owen O'Malley, Cindy Tittle, and Stephanie Aha have provided us with feedback and suggestions for features that have enabled PLEIADES to better support software engineering applications. Yidong Chen implemented PLEIADES and helped solve a number of implementation problems. Rick Hudson, Stan Sutton, and the referees provided useful feedback on an earlier draft of this paper. Finally, we would like to thank our Arcadia Consortium colleagues for years of feedback and ideas—many of the object management features PLEIADES supports resulted from their experiences.

References

- [1] S. Alagic. Persistent Metaobjects. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 31–42, Sept 1990.
- [2] A. Albano, L. Cardelli, and R. Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.
- [3] T. Andrews and C. Harris. Combining Language and Database Advances in an Object-Oriented Development Environment. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications*, pages 430–440, Oct 1987.
- [4] T. Andrews, C. Harris, and J. Duhl. The ONTOS Object Database. Product Description, 1990.
- [5] M. P. Atkinson and O.P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2):105–190, Jun 1987.
- [6] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, P.W. Cockshott, and R. Morrison. An Approach to Persistent Programming. *Computer Journal*, 26(4):360–365, Nov 1983.
- [7] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lécluse, P. Pfeffer, P. Richard, and F. Velez. The Design and Implementation of O_2 , an Object-Oriented Database Systems. In *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, pages 1–22, 1988.
- [8] N. Barghouti and G. Kaiser. Modelling Concurrency in Rule-Based Development Environments. *IEEE Expert*, 5(6), Dec 1990.
- [9] N. Barghouti and G. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, pages 269–317, Sept 1991.
- [10] P.A. Bernstein. Database System Support for Software Engineering — An Extended Abstract. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 166–179, Mar 1987.
- [11] N. Bhachech. IRIS_To_CFG User Manual. Arcadia Document UM-91-06, University of Massachusetts, 1991.
- [12] A. Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, Oct 1981.
- [13] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, H. Williams, and M. Williams. The Gemstone Data Management System. In *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F.H. Lochovsky, editors, pages 283–308. Addison-Wesley, 1988.
- [14] A.P. Buchmann, R.S. Carrera, and M.A. Vazquez-Galindo. A Generalized Constraint and Exception Handler for an Object-Oriented CAD-DBMS. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 38–49, Sept 1986.
- [15] L. Cardelli. *Amber*, pages 21–47. Springer-Verlag, 1986.
- [16] M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita. Storage Management for Objects in EXODUS. In *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F.H. Lochovsky, editors, pages 341–369. Addison-Wesley, 1988.
- [17] L.A. Clarke and D.J. Richardson and S.J. Zeil. TEAM: A Support Environment for Testing, Evaluation, and Analysis. In *Proceedings SIGSOFT '88: Third Symposium on Software Development Environment*, pages 153–162, Nov 1988.
- [18] D. Cohen. *AP5 Manual*. Univ. of Southern California, Information Sciences Institute, March 1988.

- [19] C.J. Date. *An Introduction to Database Systems*. Addison-Wesley Publishing Company, 5 edition, 1990.
- [20] U. Dayal, A. Buchmann, and D. McCarthy. Rules Are Objects Too: A Knowledge Model for an Active, Object-Oriented Database System. In *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, pages 129–143, Sept 1988.
- [21] A. Dearle, R. Connor, F. Brown, and R. Morrison. Napier88—A Database Programming Language? In *Proceedings of the Second International Workshop on Database Programming Language*, pages 179–195, Jun 1989.
- [22] S. Ford, J. Joseph, D.E. Langworthy, D.F. Lively, G. Pathak, E.R. Perez, R.W. Peterson, D.M. Sparacin, S.M. Thatte, D.L. Wells, and S. Agarwala. ZEITGEIST: Database Support for Object-Oriented Programming. In *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, pages 23–42, Sept 1988.
- [23] S. Hudson and R. King. CACTIS: A Database System for Specifying Functionally-Defined Data. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 26–37, 1986.
- [24] R. Kadia. Issues Encountered in Building a Flexible Software Development Environment: Lessons from the Arcadia Project. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments (SDE5)*, pages 169–180, Dec 1992.
- [25] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1990.
- [26] W. Kim, N. Ballou, J. Banerjee, H-T. Chou, J.F. Garza, and D. Woelk. Integrating an Object-Oriented Programming System with a Database System. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications*, 1988.
- [27] A. Koren. Identifying Type Changes in a Collection of Evolving Types. Masters project report, Computer Science Department, University of Massachusetts, Amherst, May 1992.
- [28] P. Lee, F. Pfenning, G. Rollins, and D. Scott. The Ergo Support System: An Integrated Set of Tools for Prototyping Integrated Environments. In *Proceedings of SIGSOFT '88: Third Symposium Software Development Environments*, pages 25–34, Nov 1988.
- [29] B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, B. Schiefler, and A. Snyder. *Lecture Notes in Computer Science, Vol. 114*, chapter CLU Reference Manual. Springer-Verlag, 1981.
- [30] G. Lohman, B. Lindsay, H. Pirahesh, and K. Bernhard Schiefler. Extensions to Starburst: Objects, Types, Functions, and Rules. *Communications of the ACM*, 34(10):94–109, Oct 1991.
- [31] J.P. Loyall, S.A. Mathisen, P.J. Hurley, J.S. Williamson, and L.A. Clarke. An Advanced System for the Verification and Validation of Real-Time Avionics Software. In *Proceedings of the Eleventh Digital Avionics Systems Conference*, Oct 1992.
- [32] D.C. Luckham and F.W. vonHenke. An Overview of ANNA, a Specification Language for Ada. *IEEE Software*, 2(2):9–24, March 1985.
- [33] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object-Oriented DBMS. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications*, 1986.
- [34] J. Mylopoulos, P.A. Bernstein, and H.K.T. Wong. Some Features of the Taxis Data Model. In *Proceedings of the Sixth International Conference on Very Large Databases*, 1980.
- [35] M. Nagl. A Software Development Environment Based on Graph Technology. Technical Report 87-3, Aachen University of Technology, 1987.
- [36] Object Design, Inc., Burlington, MA. *An Introduction to ObjectStore*, 1990.
- [37] Objectivity, Inc., Menlo Park, CA. *Objectivity Database System Overview*, 1990.
- [38] K.M. Olender and L.J. Osterweil. Interprocedural Static Analysis of Sequencing Constraints. *ACM Transactions on Software Engineering and Methodology*, 1(1):21–52, Jan 1992.
- [39] H.A. Podgurski and L.A. Clarke. A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance. *Transactions on Software Engineering*, 16(9):965–979, Sept 1990.
- [40] D.J. Richardson, T.O. O'Malley, C.T. Moore, and S.L. Aha. Developing and Integrating ProDAG in the Arcadia Environment. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, Dec 1992.
- [41] J.E. Richardson and M.J. Carey. Programming Constructs for Database System Implementation in EXODUS. In *Proceedings of ACM SIGMOD Conference*, 1987.

- [42] D. Rosenblum. Towards a Method of Programming with Assertions. In *Proceedings Fourteenth International Conference on Software Engineering*, May 1992.
- [43] J.W. Schmidt. Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems*, 2(3):247–261, Sept 1977.
- [44] M. Sirkin, D. Batory, and V. Singhal. Software Components in a Data Structure Precompiler. In *Proceedings Fifteenth International Conference on Software Engineering*, pages 437–446, May 1993.
- [45] J.M. Smith, S. Fox, and T. Landers. ADAPLEX: Rationale and Reference Manual. Technical Report CCA-83-8, Computer Corporation of America, Cambridge, MA, May 1983.
- [46] Special Issue on the Interface Description Language IDL, Nov 1987.
- [47] G.L. Steele. *The Definition and Implementation of a Computer Programming Language Based on Constraints*. PhD thesis, Department of Electrical Engineering and Computer Science, M.I.T., Aug 1980.
- [48] D. Stemple, A. Socorro, and T. Sheard. Formalizing Objects for Databases Using ADABTPL. In *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, pages 110–128, Sept 1988.
- [49] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
- [50] S.M. Sutton, Jr. *APPL/A: A Prototype Language for Software-Process Programming*. PhD thesis, University of Colorado, Boulder, CO, Aug 1990.
- [51] S.M. Sutton, Jr. A Flexible Consistency Model for Persistent Data in Software-Process Programming Languages. In *Implementing Persistent Object Bases - Principles and Practice*, A. Dearle, G.M. Shaw and S.B. Zdonik, editors, pages 305–318. Morgan Kaufman, 1991.
- [52] S.M. Sutton, Jr., D. Heimbigner, and L.J. Osterweil. Language Constructs for Managing Change in Process-Centered Environments. In *Proceedings of ACM SIGSOFT '90: Fourth Symposium on Software Development Environments*, pages 206–217, Dec 1990.
- [53] P. Tarr. Language Processing Toolset Prerelease Notes. Arcadia Document UM-91-01, University of Massachusetts, 1991.
- [54] P. Tarr and S.M. Sutton, Jr. Programming Heterogeneous Transactions for Software Development Environments. In *Proceedings of the Fifteenth International Conference on Software Engineering*, pages 358–369, May 1993.
- [55] P.L. Tarr, J.C. Wileden, and L.A. Clarke. Extending and Limiting PGraphite-style Persistence. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 74–86, Aug 1990.
- [56] R.N. Taylor, F.C. Belz, L.A. Clarke, L.J. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf, and M. Young. Foundations for the Arcadia Environment Architecture. In *Proceedings of SIGSOFT88: Third Symposium on Software Development Environment*, pages 1–13, Nov 1988.
- [57] United States Department of Defense, Washington DC. *Reference Manual for the Ada Programming Language*, Jan 1983. Military Standard Ada Programming Language.
- [58] S.L. Vandenberg and D.J. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 158–167, May 1991.
- [59] D.H.D. Warren and L.M. Pereira. Prolog—The Language and Its Implementation Compared to LISP. In *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, pages 109–115, Aug 1977.
- [60] D.L. Wells, J.A. Blakeley, and C.W. Thompson. Architecture of an Open Object-Oriented Database Management System. *Computer*, 25(10):74–82, Oct 1992.
- [61] J.C. Wileden, A.L. Wolf, C.D. Fisher, and P.L. Tarr. PGraphite: An Experiment in Persistent Typed Object Management. In *Proceedings Third ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 130–142, Nov 1988.
- [62] A.L. Wolf, L.A. Clarke, and J.C. Wileden. A Model of Visibility Control. *IEEE Transactions on Software Engineering*, 14(4):512–520, Apr 1988.
- [63] M. Young, R.N. Taylor, K. Forester, and D.J. Brodbeck. Integrated Concurrency Analysis in a Software Development Environment. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification*, pages 200–209, Dec 1989.