Formalization and Application of a Unifying Model for Name Management

Alan Kaplan
Jack C. Wileden
{kaplan,wileden}@cs.umass.edu

CMPSCI Technical Report 95–60
July 1995

Convergent Computing Systems Laboratory
Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

To appear in Proceedings of the Third Symposium on the Foundations of Software Engineering Washington, D.C., October, 1995

Formalization and Application of a Unifying Model for Name Management*

Alan Kaplan and Jack C. Wileden
Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003 USA

Abstract

Name management is among the most basic foundations of software engineering, since so many software engineering tools and techniques fundamentally depend upon manipulation of names and the entities that they represent. While many individual tools and techniques have included their own specialized, often elaborate and idiosyncratic, approaches to name management, no unifying models for name management currently exist. As a result, the individual approaches are frequently complex and error-prone, while attempts to integrate collections of software engineering tools or techniques are often impeded by the inconsistency and incompatibility of the name management approaches used in the individual components of the collection.

In this paper we consider some examples in which name management problems complicate software engineering activities. In particular, we focus on context control-related problems that often occur during software development. We then outline a unifying model of name management and show how its semantics can be formalized using *evolving algebras*, an operational semantics based on first-order logic. We also indicate how the model can serve as a basis for both informal and formal reasoning methods that can help in overcoming name management-related problems encountered by software developers.

* This material is based upon work sponsored by Texas Instruments Inc. under grant SRA-2837024. The content does not necessarily reflect the position or policy of Texas Instruments and no official endorsement should be inferred.

1 Introduction

Name management – how a computing system allows names to be established for objects, permits objects to be accessed using names, and controls the availability and meaning of names at any point in time – is fundamental to almost all aspects of computing. Various uses for names, and various mechanisms for supporting those uses, are found in nearly every domain of computing. Programming languages, operating systems, database systems, distributed systems and networks are all examples of domains in which diverse uses of names and a variety of name management mechanisms exist. Unfortunately, however, the mechanisms provided in any single domain tend to have various shortcomings. Moreover, when two or more domains converge, these shortcomings are often compounded by the incompatibility of the respective approaches to name management employed in the various domains, generally with problematic results.

The importance of name management is particularly evident in software engineering. Indeed, name management is among the most basic foundations of software engineering, since so many software engineering tools and techniques fundamentally depend upon manipulation of names and the entities that they represent. For example, name management is central to any tool or technique for configuration management, compilation support, specification or design, testing, reuse, module interconnection or any aspect of programming (e.g., [27]). As a result, individual tools and techniques for these and other software engineering tasks each tend to include their own approaches to name management. Since they are not generally based on any fundamental conceptual foundations for name management, however, these approaches are frequently complex and error-prone. Furthermore, because no unifying models or mechanisms for name management currently exist, efforts to integrate two or more software engineering techniques or tools are often impeded by the inconsistency and incompatibility of the name management approaches used in the individual techniques and tools.

Problems that arise during software engineering activities can often be traced to complexities or shortcomings in name management mechanisms. In the end, such problems are generally the result of one or more names not having the meaning that was expected. Sometimes a name has no meaning at all – a misspelling is a trivial example of this case. The resulting difficulties can be serious, but are usually easy to locate and rectify. More problematic are the cases where a name does have a meaning, but not the one that was anticipated. The difficulties arising from this kind of naming problem are often extremely subtle, hard to locate and hard to repair, especially in large and complex software systems.

The ubiquity of software engineering problems attributable to complexities or shortcomings in name management mechanisms has led us to embark on a broad-based investigation of these topics. This investigation has two primary facets. One facet is the development of *models* that can serve as a basis for enumerating, explicating or evaluating various approaches to various aspects of name management. The long-range goal of this facet is to provide a foundation for fundamental understanding of name management and its role in all domains of computing. The other facet of our research is the definition and implementation of, and experimentation with, prototype *mechanisms* for particular aspects of name management. Here the long-term goal is to provide a comprehensive set of powerful, flexible, uniform and broadly applicable mechanisms that will ease the construction and maintenance of (especially large and complicated) computing systems.

In this paper we begin by considering some examples in which name management problems complicate software engineering activities. In particular, we focus on context control-related problems that often occur during software development. With these examples as motivation, we then outline PICCOLO, a unifying model of name management, and show how it provides insights into, and a basis for informal reasoning about, name management mechanisms and the problems that arise in their use. Next we show how the semantics of the PICCOLO model's concepts can be formalized using *evolving algebras*, an operational semantics based on first-order logic. Finally, we demonstrate how the formalization of the model can serve as a basis for novel analysis techniques aimed at overcoming name management-related problems encountered by software developers.

2 Examples

In this section we illustrate some of the ways in which name management problems can complicate various software engineering activities. We present three simple examples, employing three distinct name management approaches found in reasonably familiar systems representing three different aspects of software development. While our three examples illustrate only a small subset of the name management problems that can arise in practice, they nevertheless provide some indication of how serious and often subtle the resulting difficulties can be.

2.1 Make

Our first example is based on an extremely simple use of *Make* [7] (specifically, the Free Software Foundation's GNU Make [21]), a well-known and widely used Unix[1] software engineering utility. Although Make is a versatile and powerful tool with many features (see [21] for details), in our example it is used, as it most often is, to automatically determine which pieces of a large program need to be recompiled, and then issue commands to recompile them. Figure 1 depicts the contents of an extremely simple Makefile that directs the compilation (in this case, using C++) of some tool as shown in the associated Unix subdirectory and file structure.

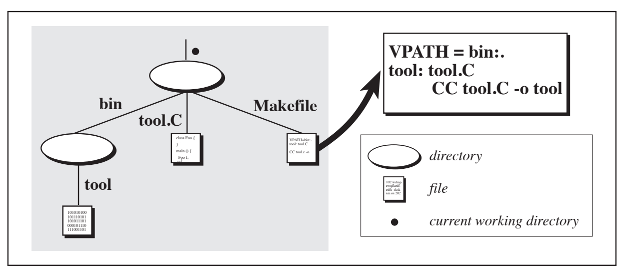

Figure 1: Makefile and Unix Directory

Make, being a tool that incorporates other tools (e.g., compilers, linkers) as components, is an example of what we call a *convergent computing system*. Hence our Make example illustrates how the inconsistencies and incompatibilities among the name management mechanisms used by different components can impede system integration. That is, while some of the name management problems in Make arise from the inadequacies of the mechanisms of the underlying components, others can be traced to incompatibilities

[1]UNIX is a registered trademark of UNIX System Laboratories, Inc.

among the various components' name management mechanisms. The root cause of the latter kind of problems is that several, often conflicting and sometimes related, mechanisms may be controlling the interpretation of names appearing on any given line(s) of the Makefile. In particular, even in our extremely simple example, different mechanisms are associated with the interpretation of names during Make's dependency and rule analysis phase and during execution of each Unix command and tool. As a result, it is often not clear what interpretation will be used for a given occurrence of a name in a Makefile.

Consider, for instance, the second line in our simple example Makefile, which defines a temporal relationship between **tool** and **tool.C**, and the next line in the Makefile, which specifies that **tool.C** is to be (re)compiled if the specified temporal relationship does not hold. If the compilation is successful, a new executable named **tool** is created. It turns out that interpretation of the occurrence of the name **tool.C** on the second line is controlled by Make's own name management mechanism based on the current contents of the **VPATH** variable (appearing on the first line of the Makefile), while interpretation of the occurrence of that same name on the third line is interpreted in a way, defined by the C++ compiler, that might be completely different. Thus, in general, it is entirely possible that Make will use one set of files in determining whether a relationship holds and a partially or completely different set of files in performing the compilation that is intended to re-establish the relationship in the case that it does not hold. Moreover, during a particular execution of Make, various tools may create newly named files or rename existing files, resulting in perhaps different and sometimes unanticipated interpretations of names occurring in a Makefile. The potential for serious and subtle difficulties is clear. Having the ability to specify that the desired interpretations of distinct occurrences of a given name are related, coupled with an analysis capability able to assess the consistency of the interpretations, could help to avoid some very subtle name management-related problems.

2.2 Library Management in Ada

Our next example illustrates some shortcomings found in management of Ada libraries. Although fairly elaborate, Ada's name management features actually provide relatively limited capabilities and as a result can lead to serious and subtle problems during construction or maintenance of Ada programs. In particular, they are demonstrably insufficient as a means for precisely and accurately specifying how names are to be interpreted across package boundaries [26]. Ada's approach to library management has similar deficiencies. While the designers of Ada recognized the need for a suitable mechanism for this purpose, they provided few specifics for its semantics:

> A single program library is implied for the compilation units of a compilation. The possible existence of different program libraries and the means by which they are named are not concerns of the language definition; they are concerns of the programming environment [6].

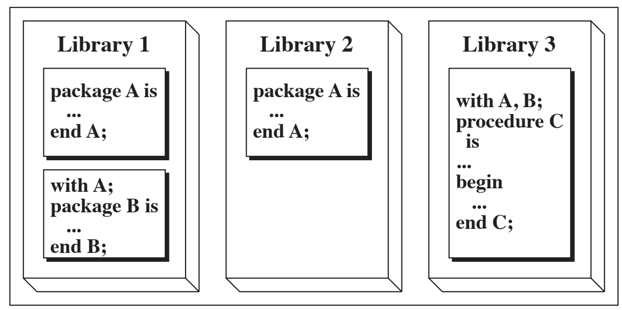

Figure 2: Ada Libraries

As an extremely simple example of some Ada library problems, consider a software system composed of a collection of modules written in Ada and organized into three separate libraries, as shown in Figure 2. In this example, procedure **C**, in **Library 3**, imports two packages, **A** and **B**. A particular client of procedure **C** might wish to have it use package **A** defined in **Library 2** and package **B** defined in **Library 1**. Notice, however, that package **B** in **Library 1** also imports a package **A**. This raises some interesting issues with respect to how individual packages specify the intended interpretations of names that they employ. For instance, should the interpretation of names **A** and **B** that is to be used by procedure **C** in **Library 3** be inherited by all other compilation units defined in all libraries? Or, can different interpretations be specified for individual libraries? Or, is a combination allowed? Unfortunately, the lack of a standard answer to these questions has resulted in vendor-specific solutions that are neither general, flexible, nor portable.

Such problems also complicate, and therefore inhibit, inclusion of packages that define overlapping sets of names, which in turn can curtail software reuse and limit the value of standard libraries. While the provision of child libraries in Ada95 [15] ameliorates such problems to some extent, in general, these and other related difficulties are further evidence of how inadequacies in name management aspects of library management can lead to software engineering problems. Having a sufficiently powerful, general and flexible conceptual foundation for rigorously specifying, assessing and comparing policies used in different library management systems would be an important step toward overcoming such problems.

2.3 Object-Oriented Databases

As a third example, we briefly consider some name management issues arising in object-oriented databases (under which heading we include persistent and database programming languages). Systems of this kind are of interest to software engineering not only because of the challenges inherent in constructing and maintaining them but also because much of their promise lies in their ability to support complex applications, including software engineering environments [25]. OODBs, however, raise a host of challenging research issues for name management. One reason for this is that they are another example of convergent computing systems, in this case representing the convergence of the programming language domain with the domains of database or operating systems. Just as other aspects of inconsistency and incompatibility among their underlying domains, such as differences in paradigms and type models, have been characterized as *impedance mismatches* in OODBs [28], we consider the existence of, and the incompatibilities among, multiple and separate name management mechanisms in their underlying domains to be yet another aspect of the impedance mismatch problem in object-oriented databases. Here we mention only three representative problems. A more extensive examination of these issues can be found in [17].

One of the primary goals of OODBs is to make *persistence* an orthogonal property to type, thus obviating the need, at least from the programmer's perspective, for traditional file systems and (relational) databases. A flexible and powerful approach to name management is crucial, therefore, since diverse applications written in different languages need to deposit and access objects in the persistent store. One shortcoming of name management mechanisms found in many current OODBs (e.g., the TI/Arpa Open Object-Oriented Database [24]) is that their persistent stores are restricted to a single, flat set of names. This results in *ad hoc* naming conventions such as suffixing users' initials to mailbox objects or prefixing vendors' names to off-the-shelf objects, in order to avoid name conflicts, and in the end unnecessarily clutters the persistent store [1]. Another problem involves the time at which name interpretation takes place. In OODBs, interpretation of names for persistent objects is often restricted to run-time. If persistence is indeed orthogonal, then OODBs should provide a range of name interpretation times matching those provided by programming languages and operating systems (e.g., composition, compile, link and load time) [19]. Finally, OODB developers and maintainers are provided with little, if any, support for determining the name reference behavior for an object. In order to determine this information, developers and maintainers face the tedious and error-prone task of manually analyzing source code, configuration scripts, etc. Thus, for example, having the ability to query an object directly about its relevant name reference behavior and having such information uniformly accessible would be a valuable software engineering aid.

3 Piccolo

As the examples in the previous section demonstrate, various tools and systems used in software engineering employ distinct, specialized, often elaborate and idiosyncratic, and sometimes multiple, approaches to name management. As a result, these tools and systems can be difficult to use, and correct usage can be hard to achieve and even harder to confirm. We believe that this situation calls for improved support for reasoning about existing name management mechanisms, and also for improved name management mechanisms for use in both applications and software tools.

An important step toward both of these objectives would be a unifying model of name management. To that end, we have developed the PICCOLO[2] model. Because of the ubiquitous nature of names in computing, there have been a number of formalisms involving or related to the use of names (e.g., [4, 8, 16, 19, 20, 22, 26]). Such formalisms, however, have generally been specific to individual, or limited sets of, computation domains. PICCOLO generalizes various concepts found in many of these models and extends them with additional features to produce a unified perspective on a more complete set of name management issues across a much broader range of computational systems.

In the remainder of this section we briefly outline the PICCOLO model and show how it can support informal reasoning about name management-related properties of software systems and tools. The next

[2]Precise Interface and Context Control in Object Libraries and Objects

section describes a formalization of the model's semantics and demonstrates its potential as a basis for formal reasoning and automated analysis tools.

3.1 Overview of the Model

The PICCOLO model is based on the following set of fundamental name management concepts:

object: An item of interest in a given setting.

name: An identifier used to reference, access or manipulate an object.

binding: In its simplest, most basic form a (name,object) pair. The availability of binding (n, o) makes it possible to use name n to reference, access or manipulate object o. Bindings may also include additional information, such as type and mutability information (as, for example, in Napier [5]).

binding space: A set of bindings that serves as a collection of definitions for names.

context: A set of bindings that is available for use in referencing, accessing or manipulating objects. A context may consist of, or be formed from, one or more binding spaces, or parts of binding spaces, or other contexts. PICCOLO's explicit distinction between binding spaces, which are primarily a means for organizing collections of bindings, typically for user convenience, and contexts, which a system uses in interpreting names during its operation, significantly facilitates modeling of many name management approaches.

closure: In its simplest, most basic form an (object, context) pair. Given the existence of the closure (o, c), it is possible for object o to use context c in referencing, accessing or manipulating other objects.

resolution: The action of returning an object when given a name and a context.

As indicated above, one shortcoming of most existing models of name management is that they do not carefully distinguish between the distinct, but clearly related, concepts of binding space and context. Another shortcoming is the typical assumption that all name management activities occur at one particular time during the lifetime of a system. This makes it difficult to model many complex situations, so the PICCOLO model provides a means for explicitly representing distinct times at which name management activities may occur:

epoch: An epoch denotes a particular time period during which name management activities may take place. The set of all epochs is modeled by an enumeration such as $E = \{e_1, e_2, \ldots\}$.

Given these fundamental concepts, the PICCOLO model defines specific name management approaches and mechanisms using the following two kinds of components:

context formation template (CFT): A collection of directives governing the formation of contexts. In a typical approach to name management we can further distinguish between two kinds of CFTs:

- **Requestor:** A CFT Requestor (or CFTR) describes the context requirements for an object. In particular, it specifies a collection of names referenced by the object. It can additionally include directives indicating such things as preferred sources of definitions (i.e., binding spaces) for the referenced names and preferred times (i.e., epochs) for context formation or modification steps.
- **Provider:** A CFT Provider (or CFTP) describes the context definitions potentially available from an object. Analogously to CFTRs, it can additionally specify such things as preferred targets (i.e., other objects) for these definitions and preferred times for context formation or modification steps.

context formation process (CFP): A procedure that produces a context from an initial context and one or more CFTs.

Applying the PICCOLO model involves defining a specific set of epochs appropriate to the host system's context formation and manipulation needs, specific kinds of directives that can appear in CFTs and one or more specific CFPs. It also implies a suitable generalization of the concept of closure, such as the following:

closure $\equiv (o, (C_i, \{CFT^*, CFP^*\}))$ where C_i represents an initial context, CFT^* represents a (possibly empty) set of context formation templates for directing the incremental formation of future contexts, and CFP^* represents a corresponding set of context formation processes that will be directed by those CFTs during incremental formation of future contexts.

3.2 Applications of the Model

We envision two major categories of use for the PICCOLO model. First, it can serve as a basis for analyzing context control mechanisms and problems. Careful description of a particular name management mechanism in terms of PICCOLO's constructs permits rigorous reasoning about and comparison of various existing, proposed or possible approaches. We give examples of this use of the model later in this section, where we use it in informal reasoning about some example name management questions, and also in the next section, where we base more formal analyses on a suitably formalized version of the PICCOLO model.

Second, the PICCOLO model can serve as a foundation for defining and implementing name management mechanisms. By defining a mechanism using the model, we can avoid *ad hoc* solutions, reason about properties of a mechanism before implementing it, and achieve uniformity in the functioning of the mechanism. We have, in fact, used PICCOLO as a basis for implementing a prototype user interface shell for an object-oriented database system [17]. Being based on the PICCOLO concepts, this shell provides much more general and powerful name management capabilities to users of the OODB than did its original interface, or than are found in most such systems.

As a simple illustration of the PICCOLO model's potential value to software developers, consider the following informal reasoning about the interpretation of names in the simple Make example of Figure 1. Guided by the model, a developer would be more likely to attempt to understand Make's name management in terms of context formation epochs, name resolution epochs, closures and which constructs in a Makefile may have closures associated with them. Pursuing this reasoning leads to several insights. One insight is that there are two separate name resolution epochs associated with the actions of this simple Makefile, one with dependency analysis and the other with execution of the compile (**CC**) command. A related observation is that each name resolution epoch has a corresponding context formation epoch. An attempt to produce even an informal description of the closure information associated with each epoch reveals that, although they use essentially the same CFP (based on traversing a search path to locate bindings) and all the CFTPs are trivial (reflecting the simplicity of access controls in the Unix file system), their respective CFTRs are quite different: the CFTR associated with the dependency analysis epoch is derived from the value of the **VPATH** variable, while the CFTR associated with the compilation is essentially a pointer to the current working directory. Pursuing this reasoning a step further reveals that the interpretations of the two occurrences of the name **tool.C** in the example Makefile will be the same, which was presumably the developer's intent. On the other hand, similar reasoning also reveals that the interpretations of the two occurrences of the name **tool** will be different, with the first referring to the existing object named **tool** in the directory (binding space) named **bin** while the second will refer to a new object that will be created in the current working directory as a side effect of the compilation. Since this would not succeed in rectifying the dependency failure that triggered the compilation, this is probably not what the developer intended. Applying even such informal reasoning, under the guidance of the concepts of the PICCOLO model, can reveal the error, assist the developer in revising the Makefile to correct it, and then serve as a basis for confirming that the revised version does indeed realize the behavior intended by the developer.

4 Formalization and Application of Piccolo

As we have attempted to demonstrate in the preceding section, the concepts of the PICCOLO model provide a useful framework for informal reasoning about name management-related issues and problems of interest to software engineers. Formalizing the semantics of the PICCOLO model's concepts, however, makes them suitable as a basis for formal reasoning, and automated tools supporting such reasoning, about name management aspects of software systems. In this section we explore some of the benefits accruing from a formalization of PICCOLO based on one particular approach to formal semantics, namely evolving algebras. We begin with a brief introduction to evolving algebras, then consider some aspects of an evolving algebra (EA) formalization of PICCOLO, including analysis opportunities arising from such a formalization.

4.1 Overview of Evolving Algebras

Evolving algebras were introduced by Gurevich as a framework for defining operational semantics [10]. The framework itself has evolved, its current definition being captured in [11]. Evolving algebras have been used in defining formal semantics for several programming languages, including C++ [23], Modula-2 [12] and Prolog [3], for other aspects of computational systems, such as data models for object-oriented databases [9], and for particular algorithms, such as the Kermit communication protocol [13] and

Lamport's bakery algorithm [2]. The formalism has also been used as a basis for proving properties of systems (e.g., [2, 13]) and has been implemented in the form of an interpreter [14] that can be used for investigating properties of EA descriptions of systems. Among the advantages of the EA approach is the ability to formulate semantic definitions at multiple levels of abstraction, so that different facets of a system can be described at their respective "natural abstraction levels" [11]. We have found this feature very useful in formalizing the semantics of PICCOLO concepts, as will be apparent in the examples of Section 4.2.

The evolving algebra formalism is based on multi-sorted first order logical structures with partial functions. The sorts are represented as *universes* of distinct atomic elements (or nullary functions). One important universe, for example, is *Boolean*, with elements *true* and *false*. A distinguished element *undef* is considered to belong to every universe, such that a function f is interpreted as being undefined at a tuple $\bar{a}$ if $f(\bar{a}) = \mathit{undef}$.

An operational semantics is defined by an abstract machine. An evolving algebra A is an abstract machine whose states consist of collections of elements from the universes included in A and functions defined on those universes. State transitions in A result from the addition or deletion of elements or modifications to the definitions of one or more functions, or both. The possible transitions of A are determined by the initial state of A and a finite collection of *transition rules*, defined recursively as follows:

- *Update rules* have the form:

  $$f(a_1, \ldots, a_n) := a_{n+1}$$

  where each a_i is a closed term (i.e., contains no free variables) and evaluates to an element of the appropriate universe. The meaning of the rule is that the value of function f on tuple $(a_1, \ldots, a_n)$ is changed to be a_{n+1} in the next state of the abstract machine.

- *Guarded rules* have the form:

  if a_0 **then** R_0
  elseif a_1 **then** R_1
  $\vdots$
  elseif a_n **then** R_n
  endif

  where each a_i is a closed term evaluating to an element of *Boolean* and each R_i is a transition rule. The meaning is that the R_j for the smallest j such that a_j evaluates to *true* is executed. If none of the a_i evaluate to *true*, then no rules are fired.

- A *Rule block* is a collection of transition rules:

  $$R_1, \ldots, R_n$$

  where all the rules in the block are interpreted as being executed simultaneously. Conflicts among the rules (i.e., updating the same function at the same step with different values) are not permitted.

- *Extension rules* have the form:

  extend U **by** $e_1, \ldots, e_n$ **with**
  $R_1, \ldots, R_n$
  endextend

  where U is a universe, $e_1, \ldots, e_n$ represent newly created elements of U, and R_i are transition rules, which (typically) use the newly created elements of the universe as values in their corresponding terms.

Given an initial state and a set of transition rules, a *run* of evolving algebra A consists of a sequence of states, S_i, resulting from successive application of all transition rules defined in A. *Static functions* are functions whose domains never change during a run, while *dynamic functions* may be updated. In addition, *external functions* are provided as means of interacting with the outside environment (e.g., I/O), where their values are determined by some oracle.

This brief introduction to evolving algebras, while sufficient for our purposes in this paper, glosses over a number of technical points and omits several interesting aspects of the formalism. The interested reader is referred to [11] for a complete treatment.

4.2 An EA/Piccolo Formulation of Name Management in Make and Unix

To help illustrate our application of evolving algebras to PICCOLO (i.e., EA/PICCOLO), we return our attention to the Make/Unix example first described in Section 2.1 and later examined in Section 3.2. In particular, we give a definition of an evolving algebra, $A_{\text{Make/Unix}}$, based on the PICCOLO model that formally describes name management in Make and Unix. Utilizing the EA formalism's support for multi-level descriptions, we first describe $A_{\text{Make/Unix}}$ in terms of some high level abstractions and then refine these abstractions in order to provide more precise semantics for context formation and context consistency in Make and Unix.

4.2.1 Formal Semantics

With respect to the Make/Unix example, files, names, directories and contexts are represented by the universes *object*, *name*, *bindingspace* and *context*, respectively. Furthermore, since directories (i.e., binding spaces) are nameable objects, $bindingspace \subseteq object$. Bindings in $A_{\text{Make/Unix}}$ are represented by a dynamic function, $LookUp(bindingspace, name) \rightarrow object$. Defining a name for a file (or a subdirectory) in a directory is modeled by performing an update to the *LookUp* function. Thus, for instance, the initial state of the file system in our example is described by the initial state, S_0, of $A_{\text{Make/Unix}}$, containing the objects b_1, b_2, o_1 and o_2, and the functions *LookUp*, *Abs* and *Bin*, that results from execution of the following:

---

extend *bindingspace* **by** b_1, b_2 **with**
 extend *object* **by** o_1, o_2 **with**
  $LookUp(b_1, n_{\text{tool.C}}) := o_1$
  $LookUp(b_1, n_{\text{bin}}) := b_2$
  $LookUp(b_2, n_{\text{tool}}) := o_2$
 endextend
 $Abs := b_1$
 $Bin := b_2$
endextend

---

where $b_i \in bindingspace$, $o_j \in object$ and $n_k \in name$. The (nullary) dynamic functions *Bin* and *Abs* (i.e., the active binding space) are provided to hold the current values for the **bin** and **current working** directories, respectively, during a particular run of $A_{\text{Make/Unix}}$.

Next, we define a universe representing epochs:

$epoch \equiv \{e_{\text{da}_{\text{cf}}}, e_{\text{da}_{\text{nr}}}, e_{\text{ia}_{\text{db}}}, e_{\text{ia}_{\text{cf}}}, e_{\text{ia}_{\text{nr}}}, e_{\text{ccc}}, e_{\text{hc}}\}$

---

where $e_{\text{da}_{\text{cf}}}$ represents a *dependency analysis/context formation* epoch, $e_{\text{da}_{\text{nr}}}$ represents a *dependency analysis/name resolution* epoch, $e_{\text{ia}_{\text{db}}}$ represents an *invoke action/define binding* epoch, $e_{\text{ia}_{\text{cf}}}$ represents an *invoke action/context formation* epoch, $e_{\text{ia}_{\text{nr}}}$ represents an *invoke action/name resolution* epoch, e_{ccc} represents a *context consistency check* epoch, and e_{hc} represents a *halt condition* epoch. Note that the epochs e_{ccc} and e_{hc} are not actually a part of Make's name management mechanism. We include them in our formulation, however, to indicate when appropriate analyses may be performed. (An example of such analyses is discussed in Section 4.2.4.) Finally, the dynamic function *CurrentEpoch* represents the current epoch in a run of $A_{\text{Make/Unix}}$, while the static function $NextEpoch(epoch) \rightarrow epoch$ computes the successor to a given epoch. In particular, S_0 contains the following definitions of *NextEpoch* and *CurrentEpoch*:

$NextEpoch\ (e_{\text{da}_{\text{cf}}}) := e_{\text{da}_{\text{nr}}}$ $NextEpoch\ (e_{\text{da}_{\text{nr}}}) := e_{\text{ccc}}$
$NextEpoch\ (e_{\text{ccc}}) := e_{\text{ia}_{\text{db}}}$ $NextEpoch\ (e_{\text{ia}_{\text{db}}}) := e_{\text{ia}_{\text{cf}}}$
$NextEpoch\ (e_{\text{ia}_{\text{cf}}}) := e_{\text{ia}_{\text{nr}}}$ $NextEpoch\ (e_{\text{ia}_{\text{nr}}}) := e_{\text{da}_{\text{cf}}}$
$CurrentEpoch := e_{\text{da}_{\text{cf}}}$

To complete our preliminary EA/PICCOLO description of name management in Make and Unix, we need to provide formal definitions of closure and name resolution. Recall that the PICCOLO definition of closure calls for an "object o to use context c in referencing, accessing or manipulating other objects." Representing closure in Make, however, is initially complicated by the fact that there are no such "objects" *per se*. Instead, contexts (and directives for forming contexts) are directly correlated with the different epochs that occur in Make. Rather than clutter $A_{\text{Make/Unix}}$ with artificial "objects," we use epochs to represent objects in a closure. Thus, to model closures and name resolution, we define the following external functions:

$ContextDirective(epoch) \rightarrow cft$
$ContextDefinedBy(epoch,\ cft) \rightarrow context$
$Resolve(context,\ name) \rightarrow object$

---

where *ContextDirective* returns the CFTs (where *cft* is defined as a universe in $A_{\text{Make/Unix}}$) at a particular epoch, *ContextDefinedBy* forms a context based on a given epoch and CFTs, and *Resolve* returns the object bound to a name in a given context. If the name is not found, *undef* is returned.

Through application of EA/PICCOLO, we can formally describe the fundamental concepts of the approach to name management found in Make/Unix at a fairly high level of abstraction. Although discerning potential name management-related problems requires some refinement of this abstraction (as we show below), it should be evident that EA/PICCOLO uniformly captures various features of name management in Make/Unix in a more precise manner than the informal use of PICCOLO presented in Section 3.2.

4.2.2 Name Management in Make/Unix

In this section, we refine $A_{\text{Make/Unix}}$, thereby moving to a lower level of abstraction, and provide more detailed EA/PICCOLO descriptions of context formation, name resolution and binding definition in $A_{\text{Make/Unix}}$. First, we assume the initial state, S_0, has been defined as described above. Next, $A_{\text{Make/Unix}}$ is augmented with a binary function, $Obj(epoch, name) \rightarrow object$, which holds values for objects required by a particular Make computation, and a unary function, $ContextIn(epoch) \rightarrow context$, which returns a context at a particular epoch. The transition rule modeling the use and manipulation of names in the Makefile shown in Figure 1 is given below:

if $\textit{CurrentEpoch} = e_{\mathrm{da}_{\mathrm{cf}}}$ **then**
$\quad \textit{ContextIn}(e_{\mathrm{da}_{\mathrm{cf}}}) :=$
$\qquad \textit{ContextDefinedBy}(e_{\mathrm{da}_{\mathrm{cf}}}, \textit{ContextDirective}(e_{\mathrm{da}_{\mathrm{cf}}}))$
$\quad \textit{CurrentEpoch} := \textit{NextEpoch}(\textit{CurrentEpoch})$
elseif $\textit{CurrentEpoch} = e_{\mathrm{da}_{\mathrm{nr}}}$ **then**
$\quad \textit{Obj}(e_{\mathrm{da}_{\mathrm{cf}}}, n_{\mathrm{tool}}) := \textit{Resolve}(\textit{ContextIn}(e_{\mathrm{da}_{\mathrm{cf}}}), n_{\mathrm{tool}})$
$\quad \textit{Obj}(e_{\mathrm{da}_{\mathrm{cf}}}, n_{\mathrm{tool.C}}) := \textit{Resolve}(\textit{ContextIn}(e_{\mathrm{da}_{\mathrm{cf}}}), n_{\mathrm{tool.C}})$
$\quad \textit{CompleteResolution}(n_{\mathrm{tool}}, n_{\mathrm{tool.C}})$
elseif $\textit{CurrentEpoch} = e_{\mathrm{ia}_{\mathrm{db}}}$ **then**
$\quad$ **extend** *object* **by** *newobj* **with**
$\qquad \textit{LookUp}(\textit{Bin}, n_{\mathrm{tool.C}}) := \textit{newobj}$
$\quad$ **endextend**
$\quad \textit{CurrentEpoch} := \textit{NextEpoch}(\textit{CurrentEpoch})$
elseif $\textit{CurrentEpoch} = e_{\mathrm{ia}_{\mathrm{cf}}}$ **then**
$\quad \textit{ContextIn}(e_{\mathrm{ia}_{\mathrm{cf}}}) :=$
$\qquad \textit{ContextDefinedBy}(e_{\mathrm{ia}_{\mathrm{cf}}}, \textit{ContextDirective}(e_{\mathrm{ia}_{\mathrm{cf}}}))$
$\quad \textit{CurrentEpoch} := \textit{NextEpoch}(\textit{CurrentEpoch})$
elseif $\textit{CurrentEpoch} = e_{\mathrm{ia}_{\mathrm{nr}}}$ **then**
$\quad \textit{Obj}(e_{\mathrm{ia}_{\mathrm{cf}}}, n_{\mathrm{tool}}) := \textit{Resolve}(\textit{ContextIn}(e_{\mathrm{ia}_{\mathrm{cf}}}), n_{\mathrm{tool}})$
$\quad \textit{Obj}(e_{\mathrm{ia}_{\mathrm{cf}}}, n_{\mathrm{tool.C}}) := \textit{Resolve}(\textit{ContextIn}(e_{\mathrm{ia}_{\mathrm{cf}}}), n_{\mathrm{tool.C}})$
$\quad \textit{CompleteResolution}(n_{\mathrm{tool}}, n_{\mathrm{tool.C}})$
endif

This transition rule defines for each epoch (except for e_{ccc} and e_{hc}, which are discussed in the next section) how the state of $A_{\mathrm{Make/Unix}}$ is changed to reflect the effects of the various name management operations. For example, when the *CurrentEpoch* equals $e_{\mathrm{da}_{\mathrm{cf}}}$, a new context is formed as dictated by the function *ContextDefinedBy* (along with the values of *ContextDirective* and *CurrentEpoch*). Similarly, when *CurrentEpoch* equals $e_{\mathrm{da}_{\mathrm{nr}}}$, the names n_{tool} and $n_{\mathrm{tool.C}}$ are resolved using the context constructed in the previous epoch. At the end of each epoch, *CurrentEpoch* is updated via an invocation of the *NextEpoch* function. (The macro *CompleteResolution* includes an invocation of *NextEpoch*, as described in Section 4.2.3.) Thus, in this refined EA/PICCOLO formulation of Make/Unix name management, we can see precisely how and when contexts are formed, names are resolved and bindings are defined.

4.2.3 Understanding Context Formation in Make/Unix

The next step in applying EA/PICCOLO to our example is to further refine the various abstractions in $A_{\mathrm{Make/Unix}}$. In particular, the functions *ContextDirective*, *ContextDefinedBy* and *Resolve* were earlier defined as external functions. Recall that this means their values are determined by an entity or oracle outside the given abstract machine. In this section, we replace these external functions with definitions of internal functions and, as a result, provide more precise semantics for context manipulation in Make/Unix.

First, we choose to represent CFTs for Make/Unix as lists of binding spaces and define a static function *ContextDirective (epoch)* $\rightarrow$ *list[bindingspace]*,[3] initialized by the execution of:

$\textit{ContextDirective}\ (e_{\mathrm{da}_{\mathrm{cf}}}) := \textit{Cons}(b_2, b_1, \textit{nil})$
$\textit{ContextDirective}\ (e_{\mathrm{ia}_{\mathrm{cf}}}) := \textit{Cons}(b_1, \textit{nil})$

The first rule corresponds to the **VPATH** specification given in the Makefile, while the second rule represents the context directives for the C++ compiler.

Next, we define *ContextDefinedBy*. In Make and Unix, contexts are evaluated lazily – when a name is resolved in a context, each binding space, in the list returned by the *ContextDirective*, is searched until either an object with the required name is found or there are no binding spaces left to search. To model this lazy evaluation approach to context formation, the context returned by *ContextDefinedBy* is simply the list of binding spaces. Thus, *ContextDefinedBy* is equated with the value of *ContextDirective*:

$\textit{ContextDefinedBy}(\textit{epoch}, \textit{ContextDirective}(\textit{epoch})) :=$
$\quad \textit{ContextDirective}(\textit{epoch})$

Finally, we define *Resolve* and *CompleteResolution* using (parameterized) macros.[4] The parameters to the *Resolve* macro are a context and a name:

Macro $\textit{obj} := \textit{Resolve}(\textit{context}, \textit{name}) \equiv$
if $\textit{ResolveStep}(\textit{name}) = \textit{SetupResolve}$ **then**
$\quad \textit{Path}(\textit{name}) := \textit{context}$
$\quad \textit{LookIn}(\textit{name}) := \textit{Car}\ (\textit{context})$
$\quad \textit{ResolveStep}(\textit{name}) := \textit{AttemptResolve}$
elseif $\textit{ResolveStep}(\textit{name}) = \textit{AttemptResolve}$ **then**
$\quad$ **if** $\textit{Path}(\textit{name}) = \textit{nil}$ **then**
$\qquad \textit{obj} := \textit{undef}$
$\qquad \textit{ResolveStep}(\textit{name}) := \textit{CompleteResolve}$
$\quad$ **elseif** $\textit{Path}(\textit{name}) \neq \textit{nil}$ **then**
$\qquad$ **if** $\textit{LookUp}(\textit{LookIn}(\textit{name}), \textit{name}) = \textit{undef}$ **then**
$\qquad\quad \textit{LookIn}(\textit{name}) := \textit{Car}(\textit{Cdr}(\textit{Path}(\textit{name})))$
$\qquad\quad \textit{Path}(\textit{name}) := \textit{Cdr}(\textit{Path}\ (\textit{name}))$
$\qquad$ **elseif** $\textit{LookUp}(\textit{LookIn}(\textit{name}), \textit{name}) \neq \textit{undef}$ **then**
$\qquad\quad \textit{obj} := \textit{LookUp}(\textit{LookIn}(\textit{name}), \textit{name})$
$\qquad\quad \textit{ResolveStep}(\textit{name}) := \textit{CompleteResolve}$
$\qquad$ **endif**
$\quad$ **endif**
endif

[3] A *list* is represented as a universe in $A_{\mathrm{Make/Unix}}$. In addition, standard list operations, such as *Cons* and *Car* (with their traditional meanings) are defined on elements in *list*.

[4] EA macros are simply a syntactic shorthand and, therefore, do not affect the formalism's computational expressiveness.

As noted above, contexts are not actually formed until a name resolution is performed. The *Resolve* macro expands into a transition rule that describes this approach to context formation. To resolve a name (for an object), the body of the rule searches each binding space, as specified by the context, until either an object is found or there are no more binding spaces left to search.

The *CompleteResolution* macro is parameterized by the names being resolved:

Macro $\mathit{CompleteResolution}(\mathit{name}_1,\ \mathit{name}_2) \equiv$
if $\mathit{ResolveStep}(\mathit{name}_1) = \mathit{CompleteResolve}\ \wedge$
  $\mathit{ResolveStep}(\mathit{name}_2) = \mathit{CompleteResolve}$ **then**
 $\mathit{ResolveStep}(\mathit{name}_1) := \mathit{SetupResolve}$
 $\mathit{ResolveStep}(\mathit{name}_2) := \mathit{SetupResolve}$
 $\mathit{CurrentEpoch} := \mathit{NextEpoch}(\mathit{CurrentEpoch})$
endif

It expands into a transition rule that synchronizes completion of the activities performed by the transition rules produced by the *Resolve* macros (which are executed simultaneously), then reinitializes the *ResolveStep* functions, and updates the *CurrentEpoch* function appropriately by invoking *NextEpoch*.

4.2.4 Formal Analyses

Recall that the *epoch* universe in $A_{\mathrm{Make/Unix}}$ included two special-purpose epochs that do not have direct correlates in the Make/Unix name management mechanism. These epochs were added in order to facilitate formal reasoning about our EA/PICCOLO model of name management in Make/Unix. In particular, they are used to enable automated analyses of the model using the EA Interpreter [14]. The epoch e_{ccc} is used to indicate when a particular analysis should be performed, and the epoch e_{hc} is used to to interrupt execution of $A_{\mathrm{Make/Unix}}$ by the EA Interpreter when an analysis reports an inconsistency. For example, suppose we wish to ensure that the set of files used during Make's dependency analysis is consistent with the set used during compilation in our example Makefile. More specifically, we want to determine whether the files bound to the names **tool** and **tool.C** during Make's dependency analysis are identical (in Unix terms, they have the same i-node value) to the files bound to the same names during compilation.

Using EA/PICCOLO, we can formally represent this assertion by augmenting $A_{\mathrm{Make/Unix}}$ with a transition rule that will be executed when the current epoch is e_{ccc}:

---

if $\mathit{CurrentEpoch} = e_{\mathrm{ccc}}$ **then**
 if $\mathit{ContextIn}\ (e_{\mathrm{ia_{cf}}}) = \mathit{undef}$ **then**
  $\mathit{CurrentEpoch} := \mathit{NextEpoch}(\mathit{CurrentEpoch})$
 elseif $\mathit{Obj}(e_{\mathrm{da_{cf}}}, n_{\mathrm{tool}}) = \mathit{Obj}(e_{\mathrm{ia_{cf}}}, n_{\mathrm{tool}})\ \wedge$
   $\mathit{Obj}(e_{\mathrm{da_{cf}}}, n_{\mathrm{tool.C}}) = \mathit{Obj}(e_{\mathrm{ia_{cf}}}, n_{\mathrm{tool.C}})$ **then**
  $\mathit{CurrentEpoch} := \mathit{NextEpoch}(\mathit{CurrentEpoch})$
 else
  $\mathit{CurrentEpoch} := e_{\mathrm{hc}}$
 endif
endif

The rule first checks that a context for the invoke action epoch has indeed been defined. If it is defined, then the consistency of the two contexts can be computed by comparing the results of the most recent resolutions of each name (which were preserved as values of the *Obj* function) and determining whether or not those resolutions returned identical objects. If they did, then the contexts are consistent; otherwise, the epoch e_{hc} is triggered and the EA Interpreter execution of $A_{\mathrm{Make/Unix}}$ is interrupted.

As a preliminary experiment in automated analysis based on EA/PICCOLO, we applied the EA Interpreter to $A_{\mathrm{Make/Unix}}$. Executing the interpreter permits a software developer to iterate over, as well as examine and query, the states of an EA model. When applied to the $A_{\mathrm{Make/Unix}}$ description of our Make/Unix example, the interpreter detects a contextual inconsistency since the name **tool** is bound to two different objects in the two different contexts. Once alerted to this situation, a software developer could rectify the problem in one of several ways. For example, the **tool** file could be moved to the current working directory, or the C++ compiler could define a new binding to **tool** in the **bin** directory. Choosing the former solution requires a change to S_0, represented by the following modification to the original extension rule:

---

extend *bindingspace* **by** b_1, b_2 **with**
 extend *object* **by** o_1, o_2 **with**
  ...
  $\mathit{LookUp}(b_1, n_{\mathrm{tool}}) := o_2$
  ...
 endextend
endextend

After this change is made, the experiment can be run again and, this time, the EA Interpreter reports no context inconsistencies.[5]

[5]The complete versions of $A_{\mathrm{Make/Unix}}$ to which the EA Interpreter was applied in these experiments are available via ftp. Contact either author for more information.

5 Summary and Conclusions

In this paper we have attempted to demonstrate the importance of name management issues to software developers, presented a unifying model of name management along with a formal semantics for its major concepts, and indicated how the model can be used as a foundation for reasoning about name management mechanisms and their use in software tools and applications.

Our EA/PICCOLO formulation of Make and Unix offers a concrete demonstration of the model's potential as a basis for formally describing and reasoning about name management in a representative example of a software engineering tool. More generally, it demonstrates that such formulations can indeed serve as a basis for formal analysis, which can aid application developers in the design, implementation and maintenance of software systems. While we have shown only one example of one relatively simple analysis, it should be evident that the approach generalizes to a variety of other useful analyses.

Although here we have focused only on how the model can support analysis of existing name management approaches and problems arising from their use, we strongly believe that new, improved mechanisms for name management are also desirable, and that our model could provide a rigorous basis on which to develop them. In fact, as noted earlier, we have already done some work along these lines. The current prototype of the CONCH user interface shell [17] for the TI/Arpa Open OODB [24] implements support for context and closure definition, including simple CFT and CFP facilities, that directly reflects the main concepts found in PICCOLO. The resulting name management mechanism is uniformly applicable from the C++ and the CLOS application programmer interfaces to Open OODB, thus contributing to multi-language interoperability [18] as well as enhanced name management. Ongoing and future work on the mechanisms facet of our name management research program includes development of similar support for PICCOLO-inspired capabilities applicable to other classes of software systems and tools, such as Make-like tools and software module library systems. As demonstrated in this paper, the PICCOLO model, particularly when formalized using evolving algebras, seems to provide a sound basis for such mechanisms. Work on this facet will also include exploration of the scalability of our analysis technique and concomitant work on specialized and optimized analysis capabilities, inspired by the general model but exploiting specific properties of particular name management mechanisms in order to improve their performance.

We are also continuing to evaluate and evolve the PICCOLO model itself as the central focus of our research program's models facet. The initial success of the existing model as a basis for explaining, reasoning about, designing, implementing and analyzing name management mechanisms across a reasonably broad range of system classes, and particularly convergent computing systems, is quite encouraging. We are, however, continuing to experiment with applying the model to additional classes of systems in an effort to further confirm, or find ways to extend, its generality and applicability. We are particularly interested in the prospect of making extended closure information in a CFT/CFP-style format a standardly available part of the interface to objects in future software systems. This would have significant potential for facilitating reuse, maintenance and interoperability. In particular, it would provide a basis for overcoming many of the name management problems associated with program libraries, such as those discussed in Section 2.2. This approach will only merit serious consideration, however, when the generality and applicability of the model have been adequately established.

Acknowledgements

The authors wish to thank Prof. Neil Immerman for initially introducing us to evolving algebras. The authors would also like to thank the Evolving Algebras research group at the University of Michigan, and especially Jim Huggins, for their assistance.

References

[1] Andrews, T. Designing linguistic interfaces to an object database or what do C++, SQL and Hell have in common? In *Fourth International Workshop on Database Programming Languages* (New York, NY, Aug–Sep 1993), pp. 3–10.

[2] Börger, E., Gurevich, Y., and Rosenzweig, D. The bakery algorithm: Yet another specification and verification. In *Specification and Validation Methods*, E. Börger, Ed. Oxford University Press, 1994.

[3] Börger, E., and Rosenzweig, D. A mathematical definition of full Prolog. In *Science of Computer Programming*. 1994.

[4] Comer, D. E., and Peterson, L. L. Understanding naming in distributed systems. *Distributed Computing 3*, 2 (May 1989), 51–60.

[5] Dearle, A., Connor, R., Brown, F., and Morrison, R. Napier88—A database programming language? In *Second International Workshop on Database Programming Languages* (June 1989), R. Hull, R. Morrison, and D. Stemple, Eds., pp. 213–229.

[6] DOD. *Reference Manual for the Ada Programming Language.* United States Department of Defense, Washington, D.C., Jan. 1983. (ANSI/ MIL–STD–1815A).

[7] Feldman, S. Make – a program for maintaining computer programs. *Software-Practice and Experience 9*, 4 (Apr. 1979), 255–265.

[8] Fraser, A. On the meaning of names in programming systems. *Communications of the ACM 14*, 6 (June 1971), 409–416.

[9] Gottlob, G., Kappel, G., and Schrefl, M. Semantics of object-oriented data models - The evolving algebra approach. In *Next Generation Information System Technology, First International East/West Database Workshop* (Kiev, USSR, Oct. 1990), J. Schmidt and A. Stogny, Eds., no. 504 in Lecture Notes In Computer Science, pp. 144–160.

[10] Gurevich, Y. Logic and the challenge of computer science. In *Current Trends in Theoretical Computer Science*, E. Börger, Ed. Computer Science Press, 1988, pp. 1–57.

[11] Gurevich, Y. Evolving algebras 1993: Lipari guide. In *Specification and Validation Methods*, E. Börger, Ed. Oxford University Press, 1994.

[12] Gurevich, Y., and Morris, J. Algebraic operational semantics and Modula-2. In *First Workshop on Computer Science Logic* (1988), E. Börger, H. K. Büning, and M. Richter, Eds., no. 329 in Lecture Notes In Computer Science, pp. 81–101.

[13] Huggins, J. Kermit: Specification and verification. In *Specification and Validation Methods*, E. Börger, Ed. Oxford University Press, 1994.

[14] Huggins, J., and Mani, R. The evolving algebra interpreter, version 2.0. University of Michigan, 1995.

[15] Intermetrics, Inc. *Ada 95 Rationale.* Cambridge, MA, Jan. 1995.

[16] Johnston, J. The contour model of block structured processes. *SIGPLAN Notices 6*, 2 (June 1971), 55–82.

[17] Kaplan, A., and Wileden, J. Conch: Experimenting with enhanced name management for persistent object systems. In *Sixth International Workshop on Persistent Object Systems* (Tarascon, Provence, France, Sept. 1994).

[18] Kaplan, A., and Wileden, J. C. PolySPIN: An architecture for polylingual object-oriented databases. Submitted.

[19] Morrison, R., Atkinson, M., Brown, A., and Dearle, A. On the classification of binding mechanisms. *Information Processing Letters 34*, 1 (Feb. 1990), 51–55.

[20] Saltzer, J. Naming and binding of objects. In *Operating Systems: An Advanced Course*, no. 60 in Lecture Notes in Computer Science. Spring-Verlag, 1978, ch. 3A, pp. 99–208.

[21] Stallman, R. M., and McGrath, R. *GNU Make: A Program For Directing Recompilation*, 0.47, for make version 3.72q beta ed. Free Software Foundation, Cambridge, MA, Nov. 1994.

[22] Stoy, J. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[23] Wallace, C. The semantics of the C++ programming language. In *Specification and Validation Methods*, E. Börger, Ed. Oxford University Press, 1994.

[24] Wells, D. L., Blakely, J. A., and Thompson, C. W. Architecture of an open object-oriented database management system. *Computer 25*, 10 (Oct. 1992), 74–82.

[25] Wileden, J. C., and Wolf, A. L. Object management technology for environments: Experiences, opportunities and risks. In *Proceedings of the International Workshop on Environments* (Sept. 1989), no. 742 in Lecture Notes in Computer Science.

[26] Wolf, A., Clarke, L., and Wileden, J. A model of visibility control. *IEEE Transactions on Software Engineering 14*, 4 (Apr. 1988), 512–520.

[27] Wolf, A., Clarke, L., and Wileden, J. The AdaPIC tool set: Supporting interface control and analysis throughout the software development process. *IEEE Transactions on Software Engineering 15*, 3 (Mar. 1989), 250–263.

[28] Zdonik, S. B., and Maier, D. Fundamentals of object-oriented databases. In *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, 1990, pp. 1–32.