The Design and Implementation of Hierarchical Software Systems with Reusable Components

DON BATORY and SEAN O'MALLEY
The University of Texas

We present a domain-independent model of hierarchical software system design and construction that is based on interchangeable software components and large-scale reuse. The model unifies the conceptualizations of two independent projects, Genesis and Avoca, that are successful examples of software component/building-block technologies and domain modeling. Building-block technologies exploit large-scale reuse, rely on open architecture software, and elevate the granularity of programming to the subsystem level. Domain modeling formalizes the similarities and differences among systems of a domain. We believe our model is a blueprint for achieving software component technologies in many domains.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol architecture*; D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.2.2 [**Software Engineering**]: Tools and Techniques—*modules and interfaces, software libraries*; D.2.7 [**Software Engineering**]: Distribution and Maintenance—*extensibility*; D.2.10 [**Software Engineering**]: Design—*Methodologies, representation*; D.2.m [**Software Engineering**]: Miscellaneous—*rapid prototyping, reusable software*

General Terms: Design, Standardization

Additional Key Words and Phrases: Domain modeling, open system architectures, reuse, software building-blocks, software design

1. INTRODUCTION

Mature engineering disciplines rely heavily on well-understood technologies that have been standardized. By purchasing off-the-shelf components, engineers can create customized systems economically by building only the parts that are application-specific. Unnecessary reinvention of technology is thereby avoided.

Contemporary software systems have been simple enough for massive technology reinvention to be economically feasible. However, as software system complexity increases, technology reinvention becomes unaffordable. There are many domains today that are technologically stable and ripe for

This research was supported in part by a grant from Texas Instruments.
Authors' address: Department of Computer Sciences, The University of Texas, Austin, TX 78712.

standardization. Certainly there will be more in the future. Many domains will concern hierarchical systems, where a progression of increasingly more sophisticated software technologies are layered upon each other.

A classical, but largely unrealized, goal of software engineering is software component (building-block) technologies. Such technologies are envisioned to exploit large-scale reuse, leverage off of open-architecture designs, and elevate the granularity of programming to the subsystem level [56]. It is believed that software component technologies can be achieved through *domain analysis*, an effort to formalize the similarities and differences among systems of a mature and well-understood domain [52].

Our interest in component technologies and domain analysis has arisen from our involvement in two independent projects: Genesis and Avoca. Genesis is the first building-blocks technology for database management systems [2–9]. Using a graphical layout editor, a customized DBMS can be specified by composing prefabricated software components. A university-quality (e.g., University Ingres) DBMS—over 70,000 lines of C—can be produced and running within twenty minutes. Avoca is a system for constructing efficient and modular network software suites using a combination of preexisting and newly created communication protocols [45–48]. Protocol suites are expressed as a graph of prefabricated protocol components. The graph is loaded into a communications kernel (the *x*kernel [36]) and executed. Genesis and Avoca are successful examples of both software component technologies and domain modeling.

When we compared Genesis and Avoca, we were amazed at the similarities in their conceptual design, organization, and implementation. We concluded that the similarities were not accidental, but were intrinsic to building-block technologies.

This paper reports our efforts to unify the conceptualizations of Genesis and Avoca. We present a domain-independent model of hierarchical software system design and construction that is based on interchangeable software components and large-scale reuse. A key feature of this model, and our most novel contribution, is recognition of the fundamental role of symmetric components in large scale reuse; these components have the unusual property that they can be composed in virtually arbitrary ways. We demonstrate the practicality of our model by using it to describe accurately the systems that we have built.

Our model is actually a metamodel of large scale system construction, which we believe can be used to define models of open architectures for many different domains. We do not present a methodology for modeling a domain in terms of the metamodel; that is far beyond the scope of this paper and is the subject of ongoing work [10].

Our work affirms and extends basic insights of many pioneers in software engineering: the software families and abstract interface concepts of Parnas [51], parameterized types of Goguen [28], hierarchical system designs of Habermann [34], object-orientation of Goldberg [28], and the frameworks concept of Deutsch [25].

We begin by explaining the superstructure of large scale systems and its relationship to the design of open-architecture software.

2. THE STRUCTURE OF LARGE SCALE SOFTWARE SYSTEMS

The structure of large scale software systems can be modeled by an elementary notation that reflects the obvious fact that systems are designed as assemblies of components and that components fit together in very specific ways. The model postulates that components are instances of types and components themselves may be parameterized. The ways in which components fit together to form systems is captured elegantly through the use of typed parameters and typed expressions. We start with a presentation of the model framework and its notation. We then demonstrate the model's generality by reviewing the domain models of Genesis of Avoca, and give some insights into the problems of contemporary software systems.

2.1 The Model Framework and Notation

Basics. A *type* is a set of values. An *abstract data type (ADT)* is a type plus operations on the values of the type. A *class* is an ADT that belongs to an inheritance lattice [21]. A *component* is a closely-knit cluster of classes that act as a unit [62].

Normally, the values of a type are simple (e.g., numbers, strings, etc.). When values become complex entities, different names (other than 'type') are generally used. For example, a set of types is called a *metatype*. A set of ADTs is called a *type class* [65] and a set of classes is a *theory* [27, 30]. We will call a set of components a *realm*. Note that metatypes, type classes, theories, and realms are themselves types.

Our model deals with components and realms.

Components. The fundamental unit of large scale software construction is the component. Every component has an interface and an implementation. Following the lead of Parnas [20, 51], the interface of a component is *anything* that is visible externally to the component. Everything else belongs to its implementation.

Every component is a member of a *realm* T, where all members of T realize exactly the same interface but in different ways. This means that members of a realm are plug-compatible and interchangeable. The interface of a realm follows directly from an object-oriented design: it is the set of one or more classes (their objects, operations, and interrelationships) that are exported by each of its members. The interface of a component, however, has additional information, such as the component's name, performance characteristics, source, and object files, etc. Thus, when we say two components share the same interface or are plug-compatible, we are referring to the realm-specific portion of their interface that they have in common. (What constitutes a component interface will be defined precisely in Section 4; for now, an informal notion will suffice. In Section 5.1, we consider ways components can be members of multiple realms; for now we assume a component is a member of precisely one realm).

Libraries and parameterized components. As a practical matter, most members (components) of a realm are never implemented; few ever get beyond the paper-design stage. Those that are define a *library*. We use the notation:

$$T = \{a1, a2, a3\}$$

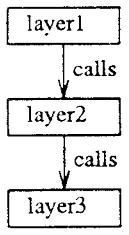
to mean that realm T has a1, a2, and a3 as library members. Realms and libraries are inherently *extensible*, as it is always possible to add another component as a member.

Components reference other components via parameters. Let the notation "t: T" mean that component t is of type T (or t belongs to realm T) and "t: {T} means t is a set of one or more members of T. Consider component c[x: R1, y: {R2}]. c has two parameters x and y, where x must be component of type R1 and y must be a set of components of type R2.

Component semantics. Every component implements an *abstract-to-concrete* mapping, which is a transformation of objects and operations visible at its interface or abstract level to objects and operations at its concrete level. As mentioned above, the abstract interface of a component is defined by its realm. The concrete interface of a component is defined by the union of the interfaces of the realms of its parameters. For example, component c[x: R1, y: {R2}]: T exports T as its abstract interface and imports R1 ∪ R2 as its concrete interface. Thus, c[ ] translates objects and operations of T to objects and operations of R1 ∪ R2, and vice versa. *A critical concept here is that components do not know how their concrete objects and operations are implemented.*

A component can be thought of as a layer, where a software system is a stacking of different layers (i.e., a composition of components). It is normally the case that components can only be composed (stacked) in a predefined order. Figure 1a shows a stacking of layers, where layer1 is on top and layer3 is on the bottom. Layer1 translates its interface objects and operations into the interface objects and operations of layer2; layer2 in turn translates its interface objects and operations into interface objects and operations of layer3. Note that if the realms of these layers (TOP, MIDDLE, and BOTTOM) are different, then only one composition of these layers is possible (Figure 1b). In Figure 1b, the libraries for TOP, MIDDLE, and BOTTOM have a single member.

Symmetric components. A distinctive and fundamental concept of our model is the possibility of *symmetric* components; i.e., components that can be composed (stacked) in arbitrary orders. More specifically, a component of realm T is symmetric iff it has at least one parameter of type T. Components d[z: R] and e[z: R] of realm R are symmetric as both have a parameter z of type R. Thus, compositions d[e[z: R]] and e[d[z: R]] are possible.



(a)

```
TOP = { layer1[ x : MIDDLE ] }

MIDDLE = { layer2[ y : BOTTOM ] }

BOTTOM = { layer3 }

LStack = layer1[ layer2[ layer3 ] ] ]
```
(b)

Fig. 1. Nonpermutable stacking of layers.

Unix file filters are prototypical examples of symmetric components. Piping the output of one filter into another is component composition: because filters have the same interface, they can be composed in different orders. Usually, the order in which components are composed makes a substantial difference in performance and semantics.

$$\text{UFILTERS} = \{\text{dtbl[x: UFILTER]}, \text{deqn[x: UFILTER]}, \ldots\}$$

Note that Unix pipe expressions like 'dtbl | deqn | ditroff' correspond to ditroff[deqn[dtbl[ ]]] in our notation.

Composition, systems, and domains. *Composition* is the rules and operations of component parameter instantiation; i.e., the guidelines by which components can be glued together. A *software system* is a type expression (i.e., a composition of components). The system LStack of Figure 1b, for example, is defined by the expression layer1[layer2[layer3]]. The set of all software systems that present the interface of realm T is called the *domain* of T, denoted Domain(T).

The concepts of domain and realm, although similar, are actually quite different. A realm is a set of components and a domain is a set of expressions. Only when a realm consists solely of parameterless components will it be indistiguishable from a domain. It is this special case that allows us to treat existing systems (e.g., commercial DBMSs) as 'primitives' in defining higher-level systems (e.g., command and control) from components.

A software tool that implements rules of composition is a component *layout editor*; it provides a language in which component expressions can be written. The set of all systems of Domain(T) that can be specified by a layout editor from compositions of library components is called the *family* of T, denoted Family(T). Family(T) is always a subset of Domain(T). Figure 2 summarizes these concepts.

Grammars and domain models. There is a close correspondence between our concepts and grammars. The left-hand side of the table below shows two

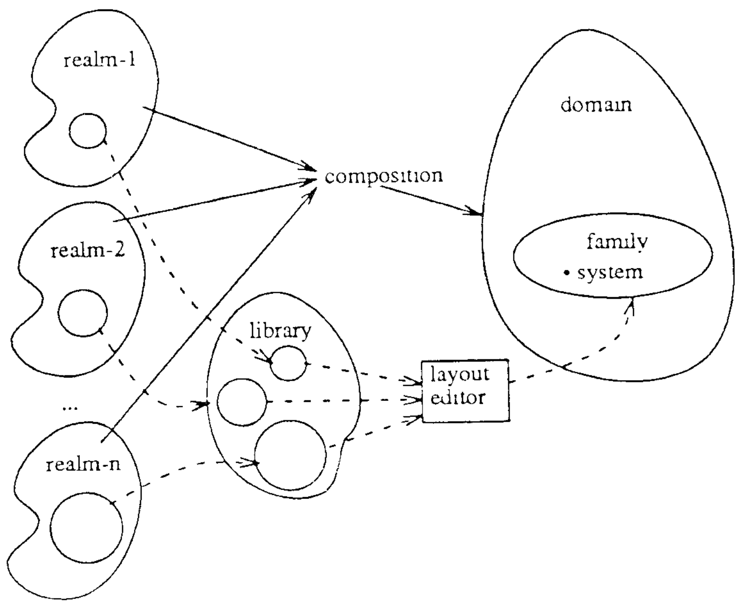


Fig. 2. Realms, libraries, composition, layout editor, family of systems, and domain.

compound productions, one for S and the other for R; the right-hand side shows the corresponding realms S and R:

$$S \rightarrow a \mid b \mid c \qquad S = \{a, b, c\},$$
$$R \rightarrow gS \mid hS \mid iR \qquad R = \{g[x{:}\ S], h[x{:}\ S], i[y{:}\ R]\}.$$

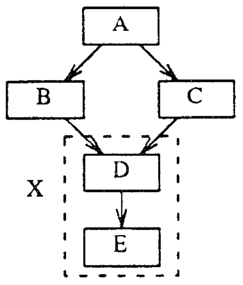
Note the following similarities. A component corresponds to a production. Parameterized components are productions whose right-hand sides reference nonterminals; parameterless components are productions that only reference terminals. Symmetric components correspond to recursive productions.

The left-hand side of a production is the interface of a component and the right-hand side is its implementation. A realm is the set of all productions with the same head. A software system is a sentence and a domain is a language. Semantic error checking is the rules of composition.

A model of a domain (or *domain model*) is the set of realms and the rules of composition that define the software systems of that domain. It is also a grammar for expressing the systems of a domain as compositions of primitive components.

Just as recursive productions play a fundamental role in compactly expressing a language, so too do symmetric components play a fundamental role in concisely expressing fundamental units of large scale reuse of a domain. We will consider examples of symmetric components shortly.

Hierarchical systems. Complex software systems are modeled as a sequence of named type expressions with no forward or recursive references. A *hierarchical* software system has an acyclic call graph where nodes are components and edges denote call relations. The hierarchical system H shown below has components B and C both calling subsystem X; X is modeled as a common subexpression that is defined separately.



H = A[B[X], C[X]]

X = D[E]

Component parameter instantiation has call-by-value semantics. System H has two instances of subsystem X; a single copy of the code exists for X but execution instances interacting with components B and C are distinct.[1]

It is worth noting that our model does admit the possibility of systems with unbounded recursion, such as Y = m[Y] which has a recursive definition. We know of no practical example of such systems; only finite replication of components (like m[m[p]]) seem to occur.

Component reuse. Recognizing and achieving software reuse are fundamental problems in software engineering. An important form of reuse is *component reuse*, which occurs in our model when two or more expressions reference the same component. Thus, if a[b[c]] and d[b[q]] are expressions (software systems), component b is reused. Common subexpressions correspond to subsystem reuse, such as subsystem X above. We will encounter two different types of reuse later: algorithm reuse and class reuse.

Examples. The domain models of Genesis and Avoca are presented in the next sections to illustrate the above concepts. Parenthetically, we note that in order to understand *any* domain model, one really needs to be familiar with the domain itself. The domain models presented below are not intended to be tutorial; citations are given so that interested readers can find additional details.

Throughout this paper, sections whose numbers are tagged with a dagger (†) marker can be skipped by readers who are not interested in concept illustrations.

2.2† Genesis: A Domain Model for Database Management Systems

The Genesis 2.0 (G2) prototype implements a portion of a domain model for database management systems. G2 enables centralized, single-client DBMSs to be synthesized from component libraries. The complete domain model,

[1] In database and network software, there may be variables or data structures within a component that may be shared by different component execution instances. A list of buffers is an example. In Avoca, semaphores are required for shared variable access; the same holds for a concurrent version of Genesis. Thus, different execution instances appear externally to be independent, internally within a component they may have a controlled interference.

described in [3–7] covers multiclient, parallel, and distributed DBMS implementations.

The model itself is nontrivial: G2 alone supports twelve distinct libraries. It is beyond the scope of this paper to cover them all or to explain the elaborate relationships (e.g., parameter instantiations) that can be admitted among components. In this section, we will highlight the terrain of the model and indicate where additional complexities lie.

2.2.2† *File Structures. Realms.* File structures map internal relations to blocks on secondary storage. They are type expressions of components from three realms: AMETHOD, NODE, and BLOCK. Some of the current G2 library members are listed below:

```
AMETHOD = {heap[d: NODE], unord[d: NODE], bplus[d, i: NODE],
           isam[d, i: NODE], hash[d: NODE], grid[d: NODE], . . . }
   NODE = {ord_prim_only[p: BLOCK], unord_prim_only[p: BLOCK],
           ord_prim_shar[p, o: BLOCK], unord_prim_shar[p, o:
           BLOCK], . . . }
  BLOCK = {fix_anch, var_anch, fix_unanch, var_unanch}
```

AMETHOD is the realm of access methods. Among library members are nonkeyed access methods (heap, unord), single-keyed methods (hash, bplus, isam), and multikeyed methods (grid). Every AMETHOD component maps an internal relation to one or two sets of logical blocks called nodes. One set of nodes contains only data records, the other set (if present) contains 'index' records. For example, bplus trees and isam files store data records in their leaf-level nodes, and store 'index' records in nonleaf nodes. Hence, the components isam[d, i: NODE] and bplus[d, i: NODE] have two parameters: (d) to specify the implementation of data/leaf nodes and (i) to specify the implementation of 'index' nodes. Some access methods, such as heap[d: NODE] and hash[d: NODE], have no index nodes and have only the single parameter (d).[2]

NODE is the realm of node implementations. A node or logical block is a sequence of records. A node component maps a logical block to one or more physical blocks, where the first block is the primary block and the remaining (if any) are overflow blocks. How primary and overflow blocks are implemented is specified by parameters (p) and (o) of NODE components. A node can optionally maintain records in key order, optionally share overflow blocks with other nodes, and optionally have primary blocks and/or overflow blocks. Each combination of options yields a distinct component. ord_prim_unshar [p, o: BLOCK], for example, maintains records in primary key order, uses a primary block, and does not share its overflow blocks with other nodes.

[2] Readers may wonder how nonkeyed, single-keyed, and multikeyed file structures can have the same interface. Recognizing that nonkeyed and single-keyed structures are degenerate cases of multikeyed structures, it follows that a general-purpose interface for multikeyed structures works for single-keyed and nonkeyed structures as well. See [5, 6, 9, 54] for further details on the implementation of Genesis.

BLOCK is the realm of record blocking methods, i.e., how records are packaged into physical blocks. Records can be optionally fixed length or variable length, and can be anchored (i.e., have physical addresses that do not change with time) or unanchored. Four components cover all options. The component fix_anch assigns anchored addresses to fixed-length records. BLOCK components do not require external services, and hence are unparameterized.

Type expressions. A file structure is an expression of type AMETHOD. An indexed sequential file (isf) that uses ordered primary unshared implementations of data nodes, and ordered primary block only implementations of index nodes, where all records are fixed-length and unanchored, corresponds to the expression:

```
isf = isam[ord_prim_unshar[fix_unanch, fix_unanch],
           ord_prim_only[fix_unanch]]
```

and an unordered file (uf) that uses unordered primary (block) only implementations of data nodes, where records are variable length and anchored, corresponds to the expression:

```
uf = unord[unord_prim_only[var_anch]]
```

Clearly, a large family of different file structures can be assembled from AMETHOD, NODE, and BLOCK components. See [6, 54] for further details.

2.2.2† *Storage Systems. Realms.* Storage systems map conceptual relations to internal relations. They are type expressions of symmetric components from the realm FMAP. Some of the current G2 library members are listed below:

```
FMAP = {index[d, i: FMAP], rl_encode[d: FMAP], zl_encode[d: FMAP],
        frag[s: FMAP], ss_bus[s: {FMAP}], internal[d: AMETHOD], ... }.
```

FMAP is the realm of file mapping components. Each transforms a conceptual file to one or more internal files. What is an 'internal' file to one component, may be a 'conceptual' file to another, and hence most FMAP components are symmetric.

Consider the fragmentation of a long record into short records that are interlinked together. Operations on long records (e.g., insertion, deletion, retrieval) have an obvious translation to operations on short records. In G2, the component frag[s: FMAP] encapsulates this mapping. It is parameterized because the method (s) by which short records are stored is unknown. Because this parameter (s) and frag[ ] are of type FMAP, frag [ ] is symmetric.

Now consider the mapping of uncompressed records to compressed records. Once again, operations on uncompressed records have an obvious translation to operations on compressed records. In G2, the components rl_encode[d: FMAP] and zl_encode[d: FMAP] encapsulate this mapping for the run-length and ziv-lempel encoding algorithms, respectively. Both are parameterized by the method of storing encoded records. As both components and their parameters are of type FMAP, both are symmetric.

The composition frag[rl_encode[d: FMAP]] corresponds to the implementation where records are fragmented before being run-length encoded. In contrast, the composition rl_encode[frag[d: FMAP]] corresponds to the implementation of encoding before fragmentation. As mentioned earlier, the order in which symmetric components are composed makes a difference both in performance and in results.

Consider two more components: index[ ] and internal[ ]. Index[d, i: FMAP] maps a conceptual file to an inverted file, which consists of a data file and one or more secondary index files. Parameter (d) specifies the implementation of the data file, and parameter (i) specifies the implementation of the index files.

Internal[d: AMETHOD] encapsulates the transformation of operations on conceptual files to low-level operations on file structures. This typically involves examining a retrieval predicate to determine how a file structure is to be searched (i.e., scanned, range retrievals, point searches, etc.).

Type expressions. A storage system is an expression of type FMAP. A storage system (ss1) that maps conceptual relations to inverted files, where data files are stored in unordered files (defined earlier by expression uf) and index files stored in isam files (defined earlier by expression isf) corresponds to the expression:

ss1 = index[internal[uf], internal[isf]]

Storage system ss2 is an enhancement of ss1 in that it handles long records. Long records are first indexed before being run-length compressed and then fragmented:

ss2 = index[rl_encode[frag[internal[uf]]], internal[isf]].

With symmetric FMAP components, a very large family of storage systems can be defined [2].

2.2.3† *Relational Database Systems. Realms.* A database is a collection of interconnected conceptual relations. A link is an interconnection between the records of two different relations. (Normally, interconnections are specified by join predicates relating records of one relation to zero or more records of another. However, relationships can be defined by manually 'connecting' one record with another. In CODASYL terminology, a link is called a set [39]).

A relational system maps a database with a nonprocedural data language interface to a set of conceptual relations (with no links) whose interface is procedural. Relational systems are compositions of components from the realms DLANG, LINK, and RSTREAM, with references to storage systems. Among the current G2 library members of these realms are:

DLANG = {sql[l: LINK, o: {OPER}], quel[l: LINK, o: {OPER}], ...}
LINK = {pointer_array[f: FMAP], ring_list[f: FMAP],
merge_join[f: FMAP], nested_loop[f: FMAP]}
RSTREAM = {cross_prod[s, s: RSTREAM], sort[s: RSTREAM], ...}

DLANG is the realm of data models and their nonprocedural data languages. Each DLANG component translates nonprocedural queries into optimized expressions that reference operations on relations and links.[3] Furthermore, each component is parameterized by (l), the method by which links between conceptual relations are implemented and (o) the set of zero or more record stream components that may be needed to process queries.

LINK is the realm of link implementations. Each component encapsulates a mapping of a database of conceptual relations and links to a database of conceptual relations without links. Every LINK component is parameterized by (f), the method of which conceptual relations are implemented.

LINK components are either 'hard' or 'soft'. 'Soft' components implement traditional relational join algorithms, like merge_join[f: FMAP] and nested_loop[f: FMAP]. 'Hard' components implement links by physically connecting records together via pointers, such as pointer_array[f: FMAP] and ring_list[f: FMAP].

RSTREAM is the realm of (usually symmetric) components that transform one or more concrete input record streams into an abstract output record stream. These components are used by DLANG components to process queries. Example components include cross product and sort.

Type expressions. A relational system is an expression of type DLANG. A relational system (rs1) that presents QUEL data model and data language, implements links by pointer arrays, and stores conceptual relations in the ss1 storage system (defined above), and references the cross product and sort components, corresponds to the expression:

rs1 = quel[pointer_array[ss1], {cross_prod[ ], sort[ ]}]

Note that the parameters of cross_prod[ ] and sort[ ] components are not instantiated. The reason is that these components are grafted onto operator trees that are generated at run-time by query optimizers. In other words, DBMSs compose certain components dynamically in order to process retrieval requests. With the exception of the RSTREAM realm, all realms that we have considered so far have components that are statically composed at design time. That RSTREAM components are composed dynamically at run time appears to be an unusual feature of DBMSs. See [7] for details on the reuse aspects of query optimization algorithms.

2.2.4† *Other Topics. Software Busses.* It is common for DBMSs to offer alternative file and/or link implementations without exposing their difference at the DBMS interface. By giving appropriate database schema directives, specific file and link implementations can be declared [5]. Genesis provides alternative implementations via multiplexing component called a *software bus*. (Software busses can be thought of as the software-lego counterpart to discriminate records). As an example, ss_bus[s: {FMAP}] is a symmetric member of the FMAP realm. It permits relations of a database to have any

[3] The DLANG interface is character input and output. quel[ ] and sql[ ] differ in the strings of characters that they recognize, even though their procedural interfaces are indistinguishable.

FMAP implementation listed in the (s) parameter of ss_bus. (Each relation of the database is tagged with an identifier which specifies how the relation should be implemented). Thus, a variation on relation system rs1 which allows relations to be stored via either storage system ss1 or ss2 is:

rs2 = quel[pointer_array[ss_bus[ss1, ss2]], {cross_prod[ ], sort[ ]}]

When a relations in a schema are declared, special statements are used to specify its implementation to be either ss1 or ss2. (The default is the first storage system listed on a bus).

Omitted parameters. Components that handle transaction management, recovery management, buffer management, primitive data types (e.g., integer, float, etc.), and other generic services (e.g., predicate evaluation) are additional parameters to the above components. We chose not to include them just to keep the model overview simple. Further details on these topics are given in [11].

Layout editors and tuning. Because type expressions quickly become difficult to read, Genesis has a layout editor, called DaTE, which enables components of different realms to be composed graphically. DaTE guarantees that design rules are not violated, which amounts to avoiding illegal compositions of components. Design rule checking is briefly considered in Section 5 and in detail in [9]. Using DaTE, a specification of a university-quality relational DBMS takes less than an half-hour. The software that is generated is *untuned* because tuning constants—that are part of every component—are assigned default values. By performing benchmarks, it is possible to tune the generated software by selectively altering tuning constants and recompiling.

2.3† Avoca: A Domain Model for Network Software Systems

Avoca is a network architecture and domain model that supports the development of encapsulated, reusable, and efficient communications protocols. Protocol suite specifications are manually written as, unlike Genesis, Avoca does not have a component layout editor.

The runtime environment for Avoca is provided by the *x*kernel: an operating system kernel designed to run network protocols [36, 45–47]. One of the goals of the *x*kernel was to support encapsulated protocols efficiently. As it turned out, most existing protocols are so unencapsulated in design and implementation, that they could not take advantage of many of the *x*kernel's features. Avoca grew out of an attempt to design protocols that could.

The domain model for Avoca has centered on the identification of realms of protocols for remote procedure calls, remote invocation methods, and network file systems. Work now underway at the University of Arizona should extend the model to cover fault tolerant protocols.

Avoca systems, or *protocol suites*, are constructed out of three types of components: Avoca protocols, virtual protocols, and existing protocols. Avoca protocols are symmetric components that can be composed in virtually arbitrary orders. Virtual protocols, a subset of Avoca protocols, dynamically multiplex over several lower-level components, and are similar to Genesis software busses. Existing protocols are generally unencapsulated and can only be composed with virtual protocols and the protocols they were originally designed to communicate with. Avoca presently has three realms: ASYNC, SYNC, and STREAM. Each is discussed in turn.

2.3.1† *Asynchronous Protocols. Realm.* Protocols that send and receive messages asynchronously are members of the ASYNC realm. Senders send messages without waiting for a reply and all messages are delivered to their destinations using upcalls [24]. If a destination replies to a message, the sender is responsible for matching requests and replies using local state or message header information. ASYNC protocols are generally the lowest level protocols supported in any network software system. Some of the current ASYNC library members are listed below, where DRIVERS, EXISTING, ASYNC_AVOCA, and VIRTUAL_ASYNC are library partitions:

$$\begin{aligned}
\text{DRIVERS} &= \{\text{amd\_eth}, \text{intel\_eth}\}\\
\text{EXISTING} &= \{\text{ip[x: ASYNC]}, \text{udp[x: ASYNC]}\}\\
\text{ASYNC\_AVOCA} &= \{\text{blast[x: ASYNC]}, \text{async\_select[x: ASYNC]}, \ldots\}\\
\text{VIRTUAL\_ASYNC} &= \{\text{vaddr[local, remote: ASYNC]}, \text{vsize[small, big: ASYNC]}\}\\
\text{ASYNC} &= \text{DRIVERS} \cup \text{EXISTING} \cup \text{ASYNC\_AVOCA} \cup\\
&\quad\ \text{VIRTUAL} \cup \text{ASYNC}
\end{aligned}$$

The ASYNC realm has a subrealm of network device drivers (DRIVERS). The Avoca library currently supports drivers for the AMD and Intel ethernet chips.

Another subrealm of ASYNC, called EXISTING, contains commonly used protocols such as ip[ ] and udp[ ]. ip[ ] is the Internet Protocol and is responsible for routing packets over the Internet and fragmenting large messages in to smaller ones [60]. udp[ ] is a simple demultiplexing protocol which delivers messages to the correct UDP port (a 16 bit integer address).

The ASYNC_AVOCA subrealm contains symmetric Avoca protocols that are not virtual. blast[ ] breaks a large message into 16 fragments, sends the fragments, and reassembles the original message once all fragments have arrived [48]. async_select[ ] is a simple demultiplexing protocol that is essentially a symmetric version of udp[ ].[4]

Virtual asynchronous protocols are found in the VIRTUAL_ASYNC subrealm. vaddr[ ] directs incoming packets to the first of its two ASYNC arguments and determines if the destination address can be reached using that component. The idea behind vaddr[ ] is a simple optimization: if the destination of a packet is on the local ethernet (the first parameter), then there is no reason to incur the overhead of transmitting it via a remote transport protocol (the second parameter).[5] vsize[ ] directs packets to the first of its two ASYNC parameters if the length of the packet is less than a predeclared size. Larger packets are directed to the component specified by the second parameter.

[4] udp[ ] explicitly uses ip addresses, and thus must sit atop ip[ ]. async_select[ ] removes this dependency.

[5] The address resolution protocol is used to make the determination of locality.

In general, virtual protocols are multiplexing components that offer the possibility of choosing between several distinct implementations on the fly and are like programming language 'if' statements where the actual path chosen for a packet is based on either static or dynamically available information. As virtual protocols are headerless, they do not affect the messages that they multiplex and hence can be composed without impacting their compatibility with EXISTING protocols. (Virtual protocols are denoted here by names beginning with the letter 'v').

Type expressions. An asynchronous protocol suite is an expression of type ASYNC. The primary purpose of Avoca was to show that encapsulated protocols can be implemented as efficiently as monolithic protocols. Hence the emphasis of Avoca is on performance. Consider the following examples.

The protocol suite neo_udp was the first to make use of virtual protocols. Common practice before Avoca was to use ip[ ] even if packets were being sent to machines in the next room. The neo_udp system (defined below) eliminated this overhead by using vaddr[ ] to bypass ip[ ] for messages to hosts on the local network (provided that these hosts also support the neo_udp optimization) and by placing these packets on the ethernet (amd_eth) directly.

neo_udp = udp[vaddr[amd_eth, ip[amd_eth]]]

Note that because vaddr[ ] does not modify the messages that it maps, neo_udp can exchange messages with systems (e.g., udp[ip[amd_eth]]) that do not take this shortcut.

As a second example, the protocol suite trans (defined below) achieves a greater level of optimization by fragmenting large packets into smaller packets. This is accomplished by using vsize[ ] to recognize long packets, and using blast[ ] for fragmentation. This optimization significantly improves the performance of most communications systems.

trans = vaddr[vsize[amd_eth, blast[amd_eth]], ip[amd_eth]]

2.3.2† *Synchronous Protocols. Realm.* Protocols that return a reply message for each message sent are members of the SYNC realm. A sender sends a message and is blocked until a reply is received. Some of the current SYNC library members are listed below:

```
EXISTING_RPC = {local_rpc, sun_rpc, sprite_rpc}
  SYNC_AVOCA = {sync_select[x: SYNC], sun_select[x: SYNC],
                 ureqrep[x: ASYNC], reqrep[x: ASYNC], . . . }
VIRTUAL_SYNC = {vrpc[local, fast_remote, slow_remote: SYNC], . . . }
        SYNC = TERMINAL ∪ EXISTING_RPC ∪ SYNC_AVOCA ∪
               VIRTUAL_SYNC
```

The most common synchronous protocol is the subrealm of remote procedure call protocols (EXISTING_RPC). It has been experimentally determined that most 'remote' procedure calls are in fact calls to different address spaces on the same machine [15]. local_rpc is an efficient protocol that relies on the local operating system to process remote procedure calls. sun_rpc is the *x*kernel implementation of Sun RPC and is fully compatible with existing implementations. sprite_rpc is the *x*kernel implementation of Sprite RPC and is again compatible with the original version.

SYNC_AVOCA is the realm of synchronous Avoca protocols. ureqrep[ ] and reqrep[ ] are protocols that perform SYNC to ASYNC conversions, and implement the request reply portion of Sun RPC and Sprite RPC respectively. ureqrep[ ] provides an unreliable request reply service while reqrep[ ] is reliable. To be reliable, a request reply protocol must deliver only one message to the destination for each request.

sync_select[ ] is the synchronous version of async_select[ ], and is used to address remote procedures [45]. sun_select[ ] is the sun demultiplexing protocol that takes three 32-bit addresses as its input, and as we'll see later, can be defined in terms of sync_select[ ]. Other SYNC_AVOCA protocols were created to perform the other functions normally bundled into RPC protocols. Discussions of these are beyond the scope of this paper.

Among the virtual synchronous protocols of subrealm VIRTUAL_SYNC, vrpc[ ] operates much like vaddr[ ] except that it supports three possible cases. If the target of the RPC is local to this machine, the protocol for parameter (local) is used. If the target is on a remote machine that supports a fast rpc that this machine understands, the protocol for parameter (fast_remote) is used. In all other cases, the protocol for parameter (slow_remote) is used.

Type expressions. Addressing in most network software systems (i.e., the assigning of an address to networking entities and the demultiplexing of incoming messages to the appropriate entity) is bundled within RPC protocols. By elevating addressing to a distinct protocol, Avoca increases the flexibility of building and modifying network software.

The protocols in the SYNC realm resulted from a decomposition of Sprite RPC and are usually configured to give Sprite RPC semantics. Sprite RPC is known to be much superior to Sun RPC and one would like to use Sprite RPC wherever possible. The easiest way to do this is to create a protocol sun_select [ ] which implements Sun RPC addressing. This allows one to create a protocol suite with Sprite RPC semantics and the Sun RPC interface. A virtual protocol vrpc[ ] can then be used which has local RPC semantics in the local case, Sprite RPC semantics whenever the destination host supports vrpc[ ], and uses Sun RPC to communicate with those machines that do not support vrpc[ ].

The following composition is a simplified version of such a system, where trans denotes the ASYNC system specified previously:

```
neo_sunrpc = vrpc[local_rpc, sun_select[reqrep[trans]], sun_rpc]
```

Note that vrpc[ ] is responsible for hiding any semantic difference between the three subsystems. The only noticeable effects are a faster and more reliable version of Sun RPC.

Now consider another example. Through repeated composition, sync_select [ ] can be used to construct more complex addressing schemes. Recall that sync_select[ ] supports a single 32 bit address while sun_select[ ] supports three 32 bit address fields. One can use sync_select[ ] to create sun_select[ ] in the following fashion:

sun_select[x: SYNC] = sync_select[sync_select[sync_select[x: SYNC]]]

The simplicity and flexibility with which addressing schemes are handled in Avoca is in stark contrast with traditional methods. In particular, the Internet community is faced with the task of modifying the internals of TCP because the 16 bit port number they use for addresses is insufficient for many modern applications [60]. In Avoca this would be a 30 minute modification.

2.3.3† *Network File Systems. Realm.* The STREAM protocol realm presents a Unix file interface, which offers file read and write operations. STREAM components offer different implementations of network file systems. Currently, the only STREAM protocol that Avoca supports is sun_nfs [ ], Sun's network file system.

STREAM = {sun_nfs[x: SYNC]}

sun_nfs[ ] is not really a communications protocol in the traditional sense, but more of a Genesis-like component. It provides a UNIX file-filter interface. We suspect that the most direct ways to merge Genesis and Avoca could be through this component/realm. One way would be to revise sun_nfs[ ] to admit components within database realms, thereby enabling different file storage schemes to be supported internally at a site. Additionally, one could revise Genesis IO-realm components, thereby enabling data to be distributed transparently across multiple hosts. We are investigating both possibilities.

Finally, we note that Avoca-produced systems do not need postproduction tuning, unlike Genesis-produced DBMSs. From our experience, protocols have been simple enough that additional tuning has not been required.

2.4 Contemporary Software Systems

Large software systems are always implemented in terms of components; they are simply too big to be designed and developed in any other way. It is possible to model existing systems as a cast-in-stone composition of handcrafted components.

The main problem with today's software are the ad hoc methods of system design and decomposition. Component reuse (which we consider synonymous with large scale reuse) is a casualty of ad hoc methods. Each of the problems below is a consequence of this casualty.

No families of systems. When contemporary software systems are examined in the context of realms, it is rare to find a realm that has a library with more than one component. One-of-a-kind systems are prime examples; their components have unique interfaces and unique implementations.[6] With the possible exception of system versions (where outdated components are re-

[6] The problem here may have a cultural aspect. It is more important today to promote a system on the basis of its uniqueness, rather than stressing its similarity with other systems. Consequently, opportunities for potential reuse are often dismissed or go unrecognized.

placed with updated ones), families of different systems do not exist. In contrast, families of multiple systems are natural by-products of component reuse.

No component interchangeability. Interchangeability of components from different systems is impossible in contemporary software. Consider the query optimizer of database systems. DBMS1[x: OPTIMIZER1,...] and DBMS2[x: OPTIMIZER2,...] are two different DBMSs, each of which has an optimizer (x) as one of its parameters. The optimizer used by DBMS1 has a different type than that for DBMS2, i.e., the optimizers of each system have *different* interfaces and are not plug compatible. Both DBMSs would need to use optimizers of the *same* type for interchangeability to be possible. Commercial DBMSs, like most software, have never been designed with interchangeability of components in mind.

No symmetric components. The 'true' building blocks for some realms are symmetric components. As we have shown, a small number of symmetric components can be composed in a vast number of ways. Not recognizing such components is a lost opportunity for achieving reuse on a large scale. Furthermore, by building only their compositions, one reduces the likelihood that a particular composition will ever be reused.

2.5 Recap

We have shown that an elementary and *domain independent* notation captures a fundamental aspect of software system design: namely, systems are assemblies of components and that components fit together in very specific ways. Our model postulates that components are instances of types and components themselves may be parameterized. Software systems are modeled elegantly as type expressions.

The type expression notation and component-realm metamodel provides a scaffolding or superstructure on which other aspects of software design can be organized. This occasionally requires concepts to be force-fitted into this framework, sometimes causing a nontrivial shift in thinking patterns in order to do so. As an example, our model suggests that the design of complex software systems is guided primarily by few decisions: (1) what components to use and (2) what is the order in which components are to be composed. The actual implementation of the components, although important, is a lower-level detail.

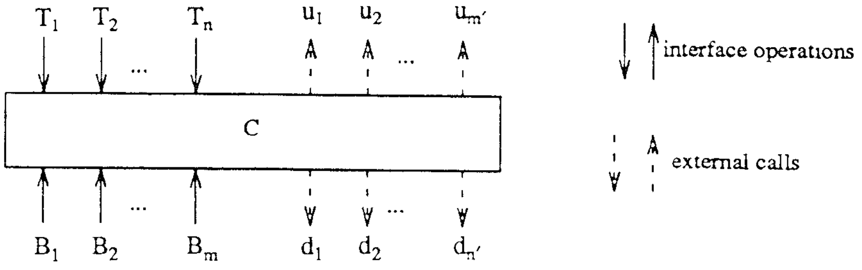
It is worth noting that over fifteen years ago Haberman observed that software system *designs* are hierarchical, but system implementations need not be [33]. Our notation and metamodel extends this insight by making the conceptual layering that exists in systems explicit. Although explicit component boundaries are maintained in Genesis- and Avoca-produced systems, this is not mandatory (e.g., components could be macroexpanded together). We amplify this perspective in the next and subsequent sections by examining features of component implementations.

3. GENERAL FEATURES OF COMPONENT IMPLEMENTATIONS

When we compared components of Genesis and Avoca, we were astonished at the similarity of their organization. On closer inspection, the commonalities that we observed would need to be present in *any* concurrent layered system. In this section, we examine general features of component implementations. A general mechanism by which concurrently executing components communicate is presented in Section 3.1. In Section 3.2, we examine how individual components can be customized, how algorithms within components fit together, and how algorithms can be reused.

3.1 A Model of Component Exteriors

Component operations. Every component C presents operations $\{T_1..T_n, B_1..B_m\}$ as its interface. Calls within C to operations external to C (whose services are provided by other components) are $\{d_1..d_{n'}, u_1..u_{m'}\}$.



The T_i operations are *top* operations, meaning that components 'above' C (i.e., components whose parameters are instantiated by C) can call T_i for lower-level services. The B_i are *bottom* operations, meaning components 'below' C (i.e., the components that instantiate C's parameters) can call B_i for higher-level services. C itself may call external operations d_i and u_i. The d_i are *down calls* requesting services from lower-level components and the u_i are *upcalls*, requests for services from higher-level components [24].

Upcalls arise in systems that receive asynchronous inputs through lower level components and that usher the processing of these inputs up through the system. Upcalls are common in networks and operating systems, and are initiated via hardware interrupts.

As a general rule, however, most layered systems process inputs strictly in a top-down manner so that bottom operations are not present. For these systems, $m = m' = 0$. Bottom operations are also absent in unparameterized components, which must be the terminal components of a system.

Component communication. Components communicate directly by calling each other's top and bottom operations. For noncurrent or nonreentrant executions, component communication is straightforward.

In the case of concurrent and reentrant executions, which is standard for networks and DBMSs, a more interesting form of communication is used. In order for several distinct executions of a component to exist simultaneously, a component must be reentrant and the state variables of each execution must be kept in a separate *control block*.

Control blocks not only provide storage for component-specific state information, they often provide the means (e.g., storage) by which data and results of computations are transmitted from one component to another. Control blocks are the primary data conduits or pipelines through which components communicate. We illustrate these ideas with examples from Genesis and Avoca shortly.

Component communications follow a standard sequence of three steps. (1) Component A initiates communication with component B by creating a control block S_B for B. This is done by calling the 'allocate-control-block' operation of B. (2) A then transmits its requests or data via calls to B, using S_B as a parameter of every call. B either writes data in S_B for A's consumption, or vice versa, depending on the operation. (3) A terminates communication by deallocating S_B, which is accomplished by calling the 'deallocate-control-block' operation of B.[7]

3.1.1† *An Example from Genesis.* Genesis components communicate through control blocks called *cursors*. Let *F* be a FMAP component and *C* be a file (FMAP) cursor. The interface to an FMAP component includes both component operations and cursor operations, some of which are listed below:

Object Operation	**Semantics**	
FMAP Component	C = make_file_cursor(F)	component F allocates a file cursor C
	first_pass(F, R)	F partially maps abstract relation definition R to its concrete counterpart
	second_pass(F, R)	F completes the mapping of R by filling in identifiers of foreign relations
FMAP Cursor	init_ret(C, Q, L)	initialize cursor C for the retrieval of records that satisfy predicate Q and returning fields specified in list L
	adv(C)	advance cursor C to next qualified record
	acc(C, A)	position cursor C over record with address A
	upd(C)	update record referenced by cursor C
	del(C)	delete record referenced by cursor C
	term_ret(C)	terminate cursor C for retrieval
	drop_file_cursor(C)	deallocate file cursor C.

A component wishing to communicate with FMAP component F: (1) allocates a file cursor *C* by calling make_file_cursor(*F*), (2) initializes *C* for retrieval via init_ret(*C*, *Q*, *L*), (3) retrieves records one at a time by calling adv(*C*), (4) terminates the retrieval via term_ret(*C*), and finally (5) deallocates *C* using

[7] A *process* can create and reference any number of control blocks. It may be the case that a control block corresponds to the state variables of a "local process" that exists only within a single component [12]. How processes generally fit into our model is a problem that we are now investigating.

drop_file_cursor(C). Allocation and initialization are separate operations as a cursor can be initialized any number of times once it is allocated.

FMAP components sit atop of AMETHOD (access method) components and communicate via file cursors. AMETHOD components, in turn, sit atop NODE (logical block) components, and communicate via node cursors. NODE components, in turn, sit atop BLOCK components and communicate via block cursors, and so on (see Figure 3a).

As a general rule, when the topmost component of a system creates a control block to a lower-level component, the lower-level component will in turn create a control block to components beneath it, and those components beneath them, and so on, causing—literally—a top-to-bottom wave of control block creations. The overhead for multiple control block creations can be reduced when the composition order of components is fixed (i.e., there is a nonpermutable stacking of layers, as in Figure 3). In such cases it is possible to define a 'composite' control block that is the union of all the control blocks that would have been allocated in a top-to-bottom creation wave. Thus, in a single and efficient operation, all needed control blocks can be allocated (or deallocated) simultaneously.

This optimization is used in Genesis. Every 'cursor' has a file (sub)cursor, a node (sub)cursor, and a block (sub)cursor inside it. When a 'cursor' is allocated/deallocated, all of its lower-level cursors are simultaneously allocated/deallocated (Figure 3b). A similar optimization is used for components that are symmetric [11].

3.1.2† *An Example from Avoca.* Avoca components communicate through control blocks called *session objects*. Components are called *protocol objects*. The interface to an ASYNC component, for example, presents a small set of operations on protocol and session objects, some of which are listed below.[8] Note that $P1$ and $P2$ denote protocol objects, S is a control block, M is a message, and A is a host address:

Object	**Operation**	**Semantics**
ASYNC Component	S = open($P1$, $P2$, A)	creates an initialized P2 session S that allows P1 to send messages to host address(s) A
	openenable($P1$, $P2$, A)	notifies P2 that P1 is willing to accept messages addressed to A
	demux($P1$, S, M)	session S performs upcall to P1 to deliver message M to P1
ASYNC Session	push(S, M)	send message M using connection S
	pop(S, M)	protocol P2 performs upcall on session S to deliver message M to S
	close(S)	protocol P1 closes connection and destroys its reference to session S

Suppose protocol P1 wants to establish a connection with some other host via protocol P2. P1 first performs an open(P1, P2, A) on P2 with the address A of

[8] It is worth noting that the interface to SYNC components is almost identical to ASYNC. The only difference is that push(S, M, M′) returns message M′ as its result.

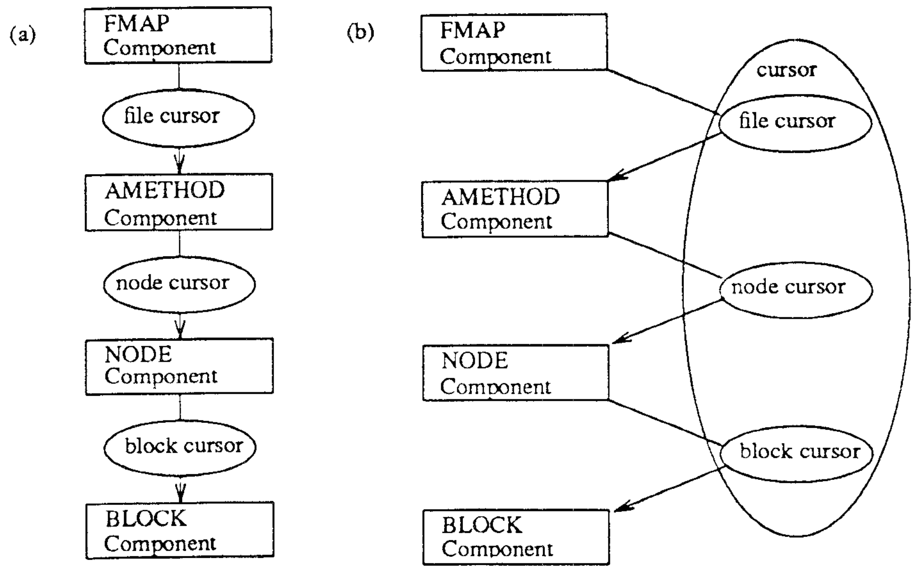


Fig. 3. Control block allocation in Genesis.

the host to create a session S. To send a message M to the host, P1 invokes the push(S, M) operation. When the remote host sends a reply to P1, the message M′ will first arrive at protocol P2 via the upcall demux(P2, S′, M′), where S′ is a session object that is delivering the incoming message to P2. As there is almost always more than one session object per protocol object, messages are demultiplexed by protocol objects to the correct session object based on the information in the message's header. Thus, P2 determines that S is the appropriate session object for M′, and makes the upcall pop(S, M′) to pass the message to S. S, in turn, passes the message to the protocol object P1 via demux(P1, S, M′), and so on.

Protocol P1 closes the connection by performing a close(S) operation. (As it is possible for multiple higher-level session objects to share a common lower-level session object, session objects are reference counted and are not always destroyed by a close( ) operation).

As a general rule, when the top-most protocol of a system creates a session to a lower-level protocol, the lower-level protocol will in turn create sessions to protocols beneath it, and those components beneath them, and so on, causing—literally—a top-to-bottom wave of session creations, identical to cursor creations in Genesis. In Avoca, the overhead for multiple session creations is not a critical concern. Most protocols create a session and use that session to send many messages, so the overhead for session creation is rarely an issue.

Figure 4 shows the protocol composition tcp[ip[ethernet]], along with created session objects. Some user (or other protocol) performed and open operation on the TCP component and received a TCP session with which it will communicate with the party whose address was passed to TCP as a

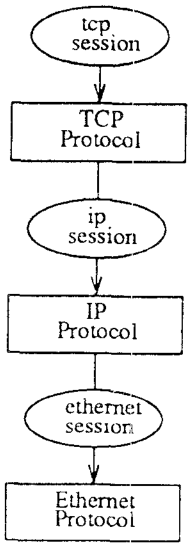


Fig. 4. Control block allocation in Avoca.

parameter to open. TCP in turn would have opened the IP component with the IP portion of the TCP address passed to it. The IP component would return a new or existing session which the TCP session uses to communicate with the remote machine. If the IP session was not reused, the IP component would have to invoke the open operation of the ethernet component and an ethernet session would be returned.

3.2 A Model of Component Interiors

Every operation of a component's interface is implemented by one, or perhaps several, algorithms. Cataloging these algorithms exposes the potentially complex internal structure of components. Algorithm catalogs explain how variations of components arise in practice, how individual components may be customized to suit a particular task, and how 'as is' algorithm reuse can be realized.

Algorithm cataloging is not appropriate for all realms. Sometimes algorithms are rarely published for security or proprietary reasons. Such is the case for command-and-control systems. Other times, algorithms within components are so intertwined that researchers and practitioners make no attempt to separate them. Such is the case with network protocols [60].

The database domain is replete with realms in which algorithm cataloging, retrieval algorithms in particular, is possible. In this section, we explain algorithm catalogs and algorithm reuse, and later illustrate their utility in the database domain.

Algorithm catalogs. There can be any number of algorithms that implement an operation of a component. Let O_C be a top or bottom operation of component C, and let S be a control block. The $j \geq 1$ algorithms that implement O_C can be cataloged as rewrite rules of the form:

$$\begin{aligned} O_C(S, arg_1, \ldots, arg_k) \Rightarrow\ & algorithm_1(S, arg_1, \ldots, arg_k, d_1(\ )..d_{n'}(\ ), u_1(\ )..u_{m'}(\ )); \\ & algorithm_2(S, arg_1, \ldots, arg_k, d_1(\ )..d_{n'}(\ ), u_1(\ )..u_{m'}(\ )); \\ & \ldots \\ & algorithm_j(S, arg_1, \ldots, arg_k, d_1(\ )..d_{n'}(\ ), u_1(\ )..u_{m'}(\ )); \end{aligned}$$

where '$\Rightarrow$' is read 'is realized by'. Every algorithm takes the arguments $(S, arg_1..arg_k)$ of its operation O_C and makes any number of down calls $d_1 \cdots d_{n'}$ and upcalls $u_1 \cdots u_{m'}$ to process its input. (Note that control block is present as a parameter if O_C is a control block operation). When components are composed, operation calls are replaced with their algorithms in the obvious way [27].[9, 10]

Rewrite rules can be adorned with additional information. Preconditions are an example. Algorithms without preconditions are *robust*, as they will always work no matter what their input might be. However, robust algorithms tend to be inefficient. Faster algorithms often exist, but work only for restricted inputs. These algorithms are *nonrobust*.

In general, robust algorithms are interchangeable, while nonrobust algorithms are not. A typical software design strategy is to use a nonrobust algorithm whenever possible (because of its performance advantages), and to use robust algorithms as a default. This means that nonrobust algorithms typically exist only in the presence of robust algorithms. We'll consider an example from Genesis shortly.

Component customization and algorithm reuse. There are many possible implementations of a component. Stated another way, a component can be customized for a particular task through an appropriate selection from catalogs of one or more algorithms for each of the component's operations. Normally, only one (robust) algorithm is selected per operation. However, it is possible for multiple algorithms to be chosen. *Algorithm reuse* occurs when the same algorithm is used in two different components. We'll see an example of algorithm reuse in the next section.

To illustrate the issues of multiple selections, query optimization in databases involves the evaluation of alternative algorithms for processing a retrieval operation. The algorithm whose preconditions are satisfied and that

[9] The elimination of explicit layering via inline expansion, which replaces an operation with its algorithmic body, was proposed in [4, 5, 33]. Inline expansion is possible in systems that have no bottom operations, because unbounded recursion may arise (e.g., a bottom call B calls a top call T, which in turn may call B again). Although recursion is definitely bounded at execution time, the amount of recursion cannot be determined at compile time. Thus, the simple idea of eliminating explicit layering via inline substitution does not always work.
[10] Most operations of a component share the same state data. By designing all operations to work off of a standard representation of state information, the compatibility of different implementations of operations is ensured.

is the cheapest (as estimated by cost functions) for a given situation is the one chosen for execution.

At first glance, one might want to include all algorithms on the presumption that the 'best' algorithm will surely be used in processing a retrieval. Generally this is a bad idea as the overhead for query optimization increases with the addition of more algorithms. From our experience, selecting few (perhaps two or three) algorithms per operation is sufficient.

3.2.1† *An Example From Genesis.* Recall that DLANG is the realm of nonprocedural data languages. Every DLANG component offers a nonprocedural retrieval statement. (For example, SQL uses the SELECT-FROM-WHERE statement; QUEL and GEM [68] use RETRIEVE-WHERE). These statements are translated into optimized expressions, called *access plans*, that are compositions of retrieval and join operations over base relations. In this section, we catalog implementations of DLANG retrieval statements. Expanded discussions on this subject are given in [3–7], and a tutorial on query optimization is presented in [37].

Let R be a relational query. (As readers will soon see, the particular language used to express R doesn't matter). The most abstract description of query processing in a relational DBMS is captured by the following rewrite which maps R to its results:

$$\text{R} \Rightarrow \text{Eval(Q\_opt(R))}$$

Q_opt( ) is the *query optimization* operation that transforms R into an executable expression E. Eval(E) executes E.

Different relational DBMSs realize Q_opt( ) in different ways. Q_opt( ) can be decomposed into a composition of three suboperations:

$$\text{Q\_opt(R)} \Rightarrow \text{Joining\_phase(Reducing\_phase(Q\_graph(R)))} \qquad (*)$$

Q_graph: R → G maps a relational query R to a query graph G [13], Reducing_phase: G → G maps a query graph without semijoin operations to one that does (which is the classical means of optimizing distributed queries [7, 37]), and Joining_phase( ): G → E maps query graphs to executable expressions. In the following paragraphs, we catalog algorithms for each of these suboperations.

*Q*_graph has many implementations, one for every data language:

```
Q_graph(R) ⇒ Sql_graph(R)       ; SQL language [22]
             Quel_graph(R)      ; QUEL language [59]
             Gem_graph(R)       ; GEM language [68]
             ...
```

The actual choice of Q_graph algorithm is component-specific (e.g., Sql_graph(*R*) is used for components that present an SQL interface, Quel_graph(*R*) is used for components that present a QUEL interface, and so on).

Reducing_phase and Joining_phase algorithms are data language independent. A partial catalog of algorithms for each operation is shown below.[11]

Reducing_phase(G) ⇒	G	; identity—no optimization
	Sdd1(G)	; SDD1 algorithm [14]
	Bc(G)	; Bernstein and Chiu algorithm [13]
	Yol(G)	; Yu et al. algorithm [69]
	. . .	
Joining_phase(G) ⇒	Sys_R(G)	; System R algorithm [57]
	U_ingres(G)	; University Ingres algorithm [66]
	Exodus(G, RS)	; Exodus rule optimizer [29]—RS is the rule set
	. . .	

From the above catalogs, it is easy to see how different implementations of components can arise. A set of potential implementations is defined by equation (*) to be the cross product of the catalogs of algorithms that implement the Q_graph, Reducing_phase, and Joining_phase operations. Just from the few that are listed, 3*4*3 = 36 distinct implementations of Q_opt can be written.

Before we consider specific combinations, we again remind readers why composition is possible. We have imposed interface standards (which includes standard data representations) on the class of relational query optimization algorithms in order to make them plug-compatible and interchangeable. This requires algorithms to be rewritten to this standard; one cannot simply copy algorithms from an existing system with ad hoc interfaces and nonstandard data representations and expect them to work.

Specific instances of (*) correspond to query processing algorithms of commercial DBMSs. The data model/data language component of DB2 [39, 57] uses the following implementation of Q_opt( ):

Q_opt(R) ⇒ Sys_R(Sql_graph(R))

That is, DB2 uses (1) Sql_graph to map SELECT-FROM-WHERE queries to query graphs, (2) the identity function G for its reducing phase algorithm, and (3) the Sys_R algorithm to map query graphs to executable expressions.

Many other combinations are possible. A DBMS with a SQL front-end that uses the University INGRES joining phase algorithm and the SDD1 reducing phase algorithm is

Q_opt(R) ⇒ U_ingres(Sdd1(Sql_graph(R)))

Another possibility is a DBMS with an SQL front-end that uses the rule-based EXODUS joining phase algorithm with the SDD1 reducing phase algorithm:

Q_opt(R) ⇒ Exodus(Sdd1(Sql_graph(R)), RS)

[11] Parameters that are not shown in these and other catalogs of this section is the list of cost functions (provided by lower-level components) that direct the optimization of G by estimating the cost of performing join, semijoin, and retrieval operations on base relations. We chose not to include them here simply because it clutters our discussion unnecessarily.

Two final points. First, recall that algorithm reuse occurs when the same algorithm is used in two different expressions. The above three Q_opt examples show the reuse of the Sql_graph algorithm. Second, it is worth noting that almost all reducing phase algorithms are nonrobust; typically reducing phase algorithms can only process tree graphs [37]. As mentioned earlier, nonrobust algorithms can be paired with robust algorithms to form new robust algorithms. A new reducing phase algorithm New_reduce(G) is a composition of the Bernstein and Chiu algorithm (which works only on tree graphs) and the identity algorithm (which works on any graph):

$$\text{New\_reduce}(G) \Rightarrow \begin{cases} \text{Bc}(G) & \text{; if G is a tree} \\ G & \text{; otherwise.} \end{cases}$$

As the above example shows, the set of Q_opt implementations is actually much larger than the cross product of the Q_graph, Reducing_phase, and Joining_phase catalogs. In analyzing existing DBMSs, one finds many implementations of Q_opt, which are cleanly modeled by algorithm catalogs. This is how we have explained variations in component implementations.

3.3 Recap

Components are not monolithic, but are suites of algorithms that translate data and operations from a component's abstract interface to data and operations of its concrete interface. When component executions are not concurrent or reentrant, components communicate by directly calling each others operations. In the case of concurrent executions, the primary parameter of an operation is a control block, which is a run-time object that contains the state variables of an execution within a component. Distinct control blocks are used to maintain the state variables of different executions of the same component.

There are many ways to implement operations of a component. We have presented a simple and informal model for cataloging the algorithms of an operation. Catalogs explain how rich variations in component implementations arise through algorithm compositions, and how 'as is' algorithm reuse is achieved.

In the next section, we explain how these ideas can be integrated with recognized concepts of object-oriented design.

4. INTEGRATION WITH OBJECT-ORIENTED DESIGN CONCEPTS

Object-oriented design concepts are necessary but not sufficient to realize large-scale reuse (i.e., component reuse). Nor are they sufficient to explain the design technology and framework that we have outlined in the previous sections.

In this section, we use the ER model to review the basic ideas of object-oriented software design. (We do not depend on a specific variant of the ER model, but only its core concepts). We then explain how our work transcends these ideas by presenting specific extensions that are needed to explain the design concepts of Genesis and Avoca. We also show how frameworks, an import OO design concept, play a role in our model. We reinforce our discussions with specific examples taken from the Genesis and Avoca prototypes.

4.1 Layered Software Designs and Transformations

Object-oriented designs. A primary result of an object-oriented design is an *object model* or *schema* that defines the classes of objects of an application and their interrelationships [19, 55, 63]. Associated with each class is the set of operations that can be performed on its objects, plus the set of object attributes.

Object models are depicted by *class diagrams*, where classes are nodes and edges are relationships. We draw class diagrams as ER diagrams, although any comparable notation will do. Figure 5 depicts three different object models—TA, TB, TC—as ER diagrams.

Layered or hierarchical designs. A *layered* or *hierarchical* software system is designed level by level. At each level, there is an object model that defines the classes, objects, and operations that the next higher level may reference. (In systems with upcalls, the next lower level may reference these operations as well). The object model at the top-most level of a system is the *external* model, and the object model at the inner-most level is the *internal* model.

Every level provides a virtual machine in which objects and operations of the next higher level (or next lower level) are defined.[12, 13, 14] For simplicity, we assume communication is always between adjacent levels.[15] Figure 6 shows (a portion of) a hierarchical design where object model TA resides on a level immediately above that of object model TB.

Transformations. A *transformation* is a mapping between adjacent levels of a system; it is the correspondence of an *abstract* or higher-level object model with a *concrete* or lower-level object model. A transformation defines

[12] It is worth noting that the ideas of hierarchical/layered system design were well-understood over two decades ago [26, 34, 50]. Astonishingly, the idea of stratified designs and layered systems is virtually absent in contemporary object-oriented literature.

[13] A primary contribution of object-orientation to the design of hierarchical systems is that an object model precisely defines both the *operations* and *objects* of a level (virtual machine). Earlier attempts did not fully recognize the role of objects that were visible at each level, thus making design encapsulations difficult (if not impossible) to achieve.

[14] Our model places no restrictions on the use of static, externally preceptible variables. The software design problem to be solved will dictate whether or not such variables are needed and can be used.

[15] Although limiting communication between adjacent levels seems restrictive, in fact it is not. If object O on level i needs to directly address an object K on level $i-j$, it is easy enough to have object models for levels $i-1$ through $i-j+1$ to include an object K' which simply transmits calls to its counterpart on the level below, eventually terminating in calls on object K on level $i-j$. As noted in [34], defining transmit calls as macros and expanding inline through a sequence of layers could possibly result in a single instruction. It is the system *design* that is hierarchical, not is implementation. Macro-expanding calls through levels is, in fact, an important implementation and modeling technique used in Genesis.

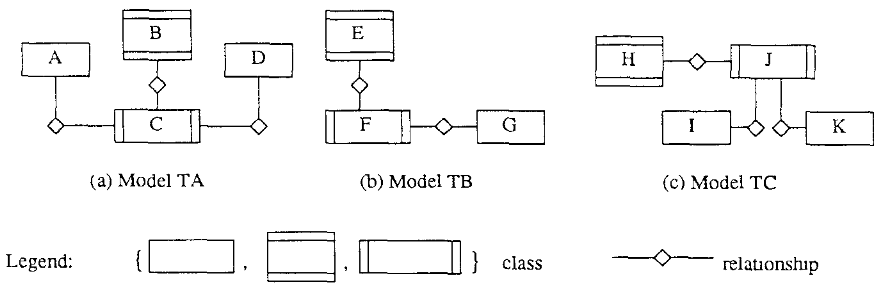


Fig. 5. Class Diagrams in ER notation.

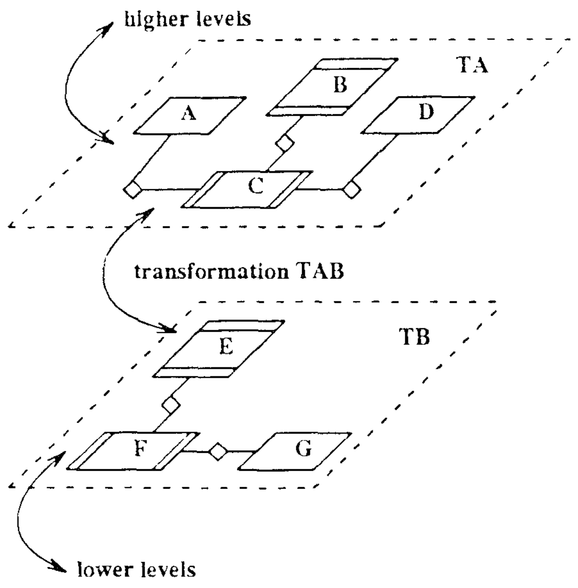


Fig. 6. Two levels of a Hierarchical System Design.

all objects and *all* operations (i.e., everything) of an abstract model in terms of objects and their operations of a concrete model. Figure 6 depicts transformation TAB, which maps abstract model TA to concrete model TB.

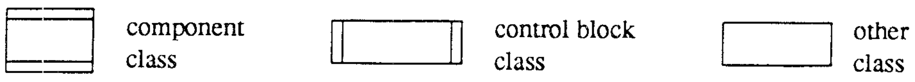
As mentioned above, hierarchical systems can be defined as a progression of increasingly more concrete object models, starting from the external model and ending with the internal model. Equivalently, a hierarchical system can be defined by (1) the external object model and (2) the 'progression' of transformations that convert the external model into the internal model. The advantage of using transformations becomes apparent when one recognizes that transformations are implemented by *components* (or *layers*). The 'upper' interface of a component is the transformation's abstract object model, and the 'lower' interface is the transformation's concrete object model. A 'progression' of transformations is a composition of components (a stacking of layers).

Realms and parameterized components. All components that implement the same object model constitute a realm. Suppose c is a component of the realm whose object model is TA. Suppose further that c transforms TA into concrete model TB, as in Figure 6. Then c has a parameter b: TB (i.e., c[b: TB]) to specify how model TB is implemented.

More generally, the concrete object model to which a component c maps may actually be the union of several disjoint concrete models, each of which has its own realm of implementing components. Every concrete model T requires c to have a parameter of type T. Figure 7 shows a component e[x: TB, y: TC] of type TA that has two parameters, one of type TB and another of type TC. (Equivalently, component $e[x, y]$ maps its abstract object model of type TA to concrete object models of types TB and TC.)

Note that Figure 7 only shows classes and relationships that belong to component interfaces. There can be additional classes and relationships hidden within component implementations that connect abstract classes to concrete classes (e.g., a relationship that connects abstract class C to concrete class H). Such details often exist, but are not shown in Figure 7.

Control blocks. Recall from Section 3.1 that components communicate directly with each other or, when there are concurrent executions, indirectly through control blocks. In a concurrent setting, a component class and *at least* one control block class must appear in every object model that defines a component interface. In Figures 5–7, we have diagrammatically distinguished component classes, control block classes, and other classes as the following.



Thus, in object model TA, class B is the component class and class C is a control block class. Other classes of an object model, in this case A and D, define additional information that is to be exchanged between components. As a general rule, operations on components (either direct or indirect) involve a translation of a web of abstract objects into a web of concrete objects. We'll illustrate examples from Genesis and Avoca shortly.

Frameworks. In object-oriented design, a class can have multiple implementations. This is accomplished by defining an *abstract* class, which only specifies the class interface, with multiple *concrete* subclasses, each providing a different implementation of the abstract class. A set of abstract classes with their concrete classes defines a *framework*, which is an important organizational concept in the design of families of software systems [25, 38].

Object models that define realm interfaces are implemented by frameworks. In the most general case, every class of an object model is an abstract class. Each component is implemented by a special concrete class for each abstract class in the model. The number of concrete classes per abstract class equals the number of components in a library. Thus, if there are A abstract classes in a realm R's object model, and there are N components in R's

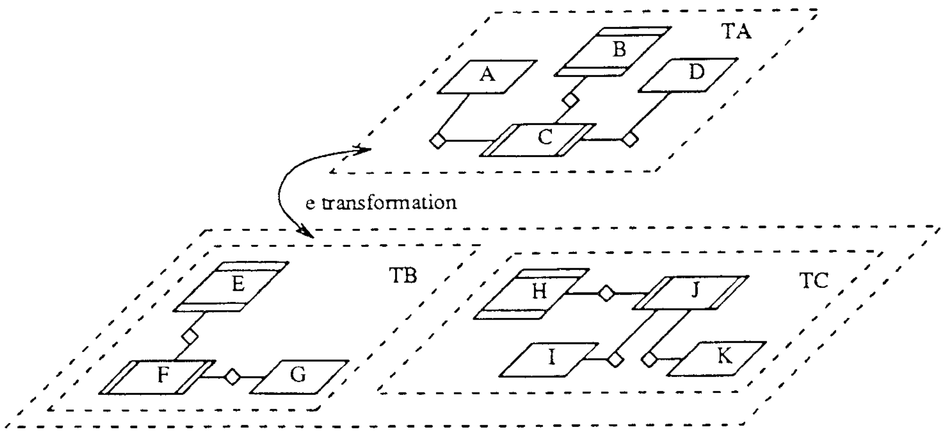


Fig. 7. Parameterized component e[x: TB, y: TC]: TA.

library, then each of the A abstract classes will have N distinct concrete subclasses, one for each component of realm R.

More typically, not all classes of an object model are abstract. Some classes have a single implementation that is shared (reused) across many different components. (This is an example of 'as is' *class reuse*). Frameworks in Genesis and Avoca fit this more common situation.

4.2† An Example From Genesis: The FMAP Interface

Recall from Section 2.2.2 that FMAP is the realm of file mapping components. Each transforms an 'abstract' file into one or more 'concrete' files. Consequently, FMAP components are symmetric.

Object model. Figure 8 shows the object model interface of FMAP components.

Instances of the *Fmap* class correspond to FMAP components. Among Fmap operations are component initialization, schema mapping, and the creation and linking of cursors (via link *RO*). Cursors are the control blocks of FMAP components.

When a database schema is compiled, there is an object in the *Relation* class for every relation. Further, each relation type is defined by a set of attributes (i.e., data fields), where each attribute is an object of the *Attribute* class. The link between relation objects and their attribute objects is *R4*.

Every cursor is an object of the *Cursor* class and is bound to single relation (*R1*). Every cursor is also bound to a retrieval predicate, called a *Query* object (*R2*), and an attribute projection list, called an *Into_List* object (*R3*). A query specifies the set of tuples of the designated relation that the cursor may reference, and the into-list specifies the attribute values of these tuples that are to be retrieved and/or modified.

A Query object is a tree of *Clause* objects, where a clause is a boolean connective or a terminal of the form '(attribute rel-op value)'. A tree of clauses

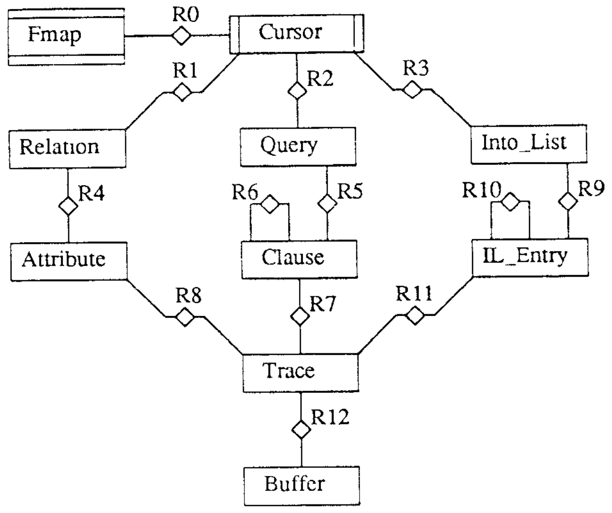


Fig. 8. The FMAP object model interface.

is specified by links *R5* and *R6*. Since a tuple may be variable-length, a special object is needed at run-time to reference the offset of an attribute in a particular tuple. Such objects are called *traces*. Every terminal clause references a Trace object (*R7*). A Trace object, in turn, references its corresponding attribute definition (*R8*) and a *buffer* in which a value of that attribute can be placed (*R12*).

An into-list object is simply a list of into-list entries (*IL-entrys*), each of which identifies an attribute that is to be retrieved and/or modified. A list of IL_entries is achieved by links *R9* and *R10*. Traces are used by intolist entries to reference attributes (*R11*). (Note that the object model suggests that traces (and their buffers) can be shared by IL_entry and Clause objects. In fact, this is commonly done).

Transformation. The object model of Figure 8 serves both as the abstract and concrete interface of symmetric FMAP components. Consider the rl_encode[d: FMAP] component which maps abstract relations (files) to compressed concrete relations. Let C be a cursor, Q be a query and IL be an into-list. (Remember that Q is connected to a web of Clause, Trace, and Buffer objects that define a selection predicate, and IL is connected to a web of IL_Entry, Trace, and Buffer objects that define an into-list).

When rl_encode receives the operation init_cursor (C, Q, IL), it initializes C to range over all tuples that satisfy predicate Q and that IL is the list of attributes for qualified records whose values are to be retrieved and/or modified. The initialization is accomplished by mapping C, Q, and IL to their concrete counterparts C′, Q′, and IL′. The translation is simple: C′ is bound to the corresponding compressed relation, Q′ is null, and IL′ references the compressed byte string that contains the compressed image of the tuple. Both Q′ and IL′ are webs of objects.

Once initialization has taken place, abstract tuples can be retrieved via the operation adv(C). A retrieval of an abstract tuple translates into the retrieval of compressed concrete tuple via adv(C′), a decoding, application of predicate Q to the decoded tuple, and if Q is satisfied, the unencoded attributes of IL are returned.

Framework notes. All classes in Figure 8 with the exception of Buffer, Trace, Query, and Into_list, are abstract. Thus, these mentioned classes have a single implementation that is shared (reused) by all FMAP components.

4.3† An Example From Avoca: The ASYNC Interface

Recall from Section 2.3 that ASYNC is the realm of asynchronous protocols that transform abstract messages into one or more concrete messages. Generally, ASYNC components are symmetric.

Object model. Figure 9 shows the object model interface to ASYNC components.

Protocol objects are instances of the *Protocol* class. Protocol objects create *Session* objects, which are the control blocks of ASYNC protocols, and interconnect them via link *S1*. This connection is needed to process the bottom operation/upcall demux( ), which routes incoming messages to the appropriate session. (Routing is accomplished by looking at the message header and extracting a protocol-specified key that identifies the recipient session). A Session object may be linked to one or more packets (via *S2*), which are instances of the *Message* class.

A Message object consists of headers and data (*S3* and *S4*). Most traditional protocol implementations support a notion of header encapsulation in which the header of a lower-level protocol is considered to be indistinguishable from message data. A key point of Avoca is that for both efficiency and software engineering reasons, segregating header and data information in any given message is a good idea. The addition and removal of message headers obeys a stack discipline and therefore message headers can be stored efficiently in a *Header_Stack* object using traditional stack operations. Message trailers, information added to the tail of a message, are rare and can be safely ignored.

The most common manipulation of message data is to fragment large messages into smaller messages and to reassemble the fragments into a single message. Therefore message data is stored in one or more *Buffer* objects that are linked together as a directed-acyclic graph (*S5*). The operations required to support typical message data manipulations are essentially identical to those of standard string manipulation. Note that the object model suggests that different message objects could share the same buffers and header stacks. Such sharing is actually critical to the efficiency of protocol code, as each copy made of a long message can easily halve the performance of a protocol suite.

The Message class, in effect, is a message abstraction that supports both stack-based header manipulation and string-based data manipulation. These

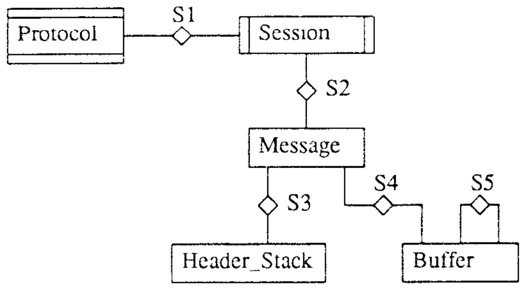


Fig. 9. The ASYNC object model interface.

abstractions sometimes interfere with one another. For example, when a large message is fragmented, the first fragment uses the original header while subsequent fragments must have an empty header stack created for them. Also in traditional protocols such as IP, data is sometimes found in the header, and headers are sometimes found in the buffer DAG. Therefore, it is important that one be able to move information between header stack and buffer representations efficiently.

Transformation. The object model of Figure 9 is both the abstract and concrete interface of symmetric ASYNC components. Consider the blast[x: ASYNC] component which fragments large messages. Let S be a session object, and let M be a Message object, which is connected to a Header_Stack object and a web of Buffer objects.

When the blast protocol receives the operation push(S, M), which means send message M to the destination of S, concrete message objects are created that represent fragments of M. For efficiency, buffers of M are shared with their message fragments. As stated above, new header stacks are created for each fragment message except for the first. A blast header is added to each message and a copy of the resulting message is saved for possible retransmission. Each concrete message is sent via session S′ that is the concrete counterpart of S.

The peer blast component receives each concrete message via a demux( ) upcall and examines its header to determine the correct session for reassembly, and then invokes that session's pop( ) operation. The selected session assembles the transmitted concrete messages by stripping the blast header from each message fragment and concatenating fragments together. Once the entire message has been formed, the session passes the message to the next protocol layer (a pointer to which is stored in every sessions local state) by invoking that protocol's demux( ) operation. These steps are repeated for each layer in the protocol hierarchy.

Framework notes. Only the Protocol and Session classes in Figure 9 are abstract. The Message, Map, Buffer, and Header_Stack classes have single implementations that are used by all ASYNC components.

4.4 Recap

Hierarchical or layered systems are designed level-by-level. Each level is designed independently of adjacent levels, and is defined by an object model whose classes, objects, and operations comprise that level's virtual machine.

Components are mappings (transformations) between adjacent levels. Components that implement the same object model belong to the same realm. Because their interfaces are the same, components within a realm are plug-compatible and interchangeable.

Contemporary object-oriented design techniques are basically two-dimensional; they concentrate on developing a single object model which can be drawn on a flat (two-dimensional) surface that explains an application. Our work recognizes and extends concepts of early software engineering pioneers [26, 34, 50] that software design techniques must be extended by another dimension in order to account for layering in hierarchical systems. Where our work departs from known results are the concepts of realms, type-expressions, and the role of interface standardization.

5. OTHER ISSUES AND RELATED WORK

The scope of this paper precludes us from addressing many of the issues that are of concern to software engineers. In this section, we briefly discuss a variety of topics that were not considered previously, but are important for understanding our results.

5.1 Model Limitations

Ad hoc layering. We believe our model is generally applicable to a broad class of hierarchical software systems. However, there may be many systems with ad hoc layering that are not expressible. In such systems, "components" result from mixtures of functional and object-oriented decompositions and do not conform to the definition of a component in our model. We are not interested in capturing ad hoc layering, nor do we believe our model can be used to do so. Our research concentrates on a subset of layered systems for which layer/component composability is possible. We have outlined properties of these systems, and without these properties, we fail to see how layer composability will work. Much of our work in domain modeling, incidentally, has been to redraw layer boundaries in ad hoc system designs so that our resulting layers/components are in fact composable. Thus we feel that our model can capture system functionality, but not ad hoc layering.

Inheritance. Our model imposes the constraint that all components of a realm must have exactly the same interface and that a component belongs to precisely one realm. This may be too restrictive and that an inheritance lattice which relates OOVM interfaces (and their realms) may be needed. For example, suppose the OOVM of realm R1 is a specialization of the OOVM for realm R0. (That is, R1 presents additional functions, possibly even additional classes, as part of its OOVM interface). Then certainly, a component of R1 could be used where ever a component of R0 is used. Such an addition, we believe, would allow us to express the class of hierarchical systems described by Habermann et al. [34].

Multiple standards. Consensus is needed for interface standards. As both of our projects relied on a small group of designers, consensus wasn't difficult. In general, however, we would expect independent design teams would arrive at different "standards." Consensus and component interoperability would then become problematic.

Even if there is general agreement on the operations that need to be supported, there can be a choice among internal data representations to use. It is certainly possible for different representations to impact performance, and choosing among alternative representations should be one of the design decisions in specifying a target system.

Other features. Our model does not explicitly show how processes fit into a building-block framework. Nor does our model explain how exceptions are handled, how distributed systems impact reusable software designs, or how parallelism can be exploited. Avoca does not compose components dynamically, but Genesis does (during query processing). Our model is geared to the static composition of components, and this may be unnecessarily restrictive. Furthermore, all components in Genesis and Avoca are hand-crafted. It is clear from our experience that there is an enormous similarity among components within a realm. This raises the possibility that software generators could be created to simplify or automate the development of large numbers of components. Thus, the distinction we have made between realms and their libraries may have limited applicability. All of these topics are subject for further research.

5.2 Related Research

Modular DBMS software. The first layered architecture to gain popularity in databases was the three-level ANSI/SPARC model [64]. Mappings between levels (the external, conceptual, and internal) were layers, whose implementation it was claimed could be altered independently. To our knowledge, no DBMS ever implemented the ANSI/SPARC architecture; it was a paper design that gave few clues about how to build actual database systems. Furthermore, ANSI/SPARC layers were effectively monolithic, leaving little opportunity for exploring and achieving component reuse.

More recently, research on extensible database systems has lead to more modular DBMS architectures. However, only Genesis and Starburst [42] were based on formal models of DBMS software. The Starburst model deals with rule-based query optimization, which essentially is a model of what we have called algorithm reuse. The optimizer "glues" algorithms together to form access plans. However, there is no notion of components, layers, or classes in the model itself. How algorithms are defined, how they are related (if at all) to components, is unclear. Starburst supports 'attachments' that are similar to, but less powerful than Genesis components [40]. Attachments are not part of the optimizer model.

Modular network software. Outside of the networking community, the ISO 7 layer model [69] is often cited as an example of the application of modular design to production systems. Inside the networking community, the ISO model is the root cause of the rejection of layering as an efficient way of implementing network software by a variety of network architectures (VMPT [23], Sprite [49]). ISO architecture sacrificed efficiency for modularity at a time when the introduction of high speed networks required an increase in efficiency. While the networking continued to use layered models to describe network software, layered implementations were actually considered harmful [61]. In [46], it was shown that it is possible to increase performance without sacrificing modularity. Some of the methods used by Avoca to organize heavily layered protocol suites were also used in the higher layers of the ISO architecture. However Avoca's approach is much more integrated with the protocol design and also more efficient.

The only modular networking architecture that is actually in use is Unix System V streams. Streams are an extension of the Unix file filter model into the kernel designed to support serial line drivers. It has been extended to support some of the implementation of communications protocols. However, how one does this is not at all obvious. In addition the central purpose of Avoca was to design protocols that compose. No attempt has ever been made to design composable stream-oriented protocols.

Module interconnection languages. MILs were proposed as a means by which previously existing modules, written in potentially different languages and running on different machines could be part of a reuse repository. Our model can be recognized as a very simple MIL. However, there are three important differences.

First, a problem with existing MIL's is that they are not sufficient to guarantee module composability, and in some cases don't even seem to encourage the creation of composable modules. Avoca is most closely related to the MIL IDL [58] and the multilanguage programming environments MLP [35] and Polylith [53]. (In fact, Polylith notation was used initially for Avoca). However the critical feature of Avoca is the methodology used to design composable protocols [47]. The notation used to describe such protocols is not nearly as important as properties such as symmetric components.

Second, by making it simple to define arbitrarily complex interfaces, the probability of any module being composable is reduced. It is not uncommon in networking for the implementor to unnecessarily expose all internal details of a protocol in the interface thus ensuring noncomposability. Third, current MILs are slow for a variety of reasons but primarily because they have sophisticated type systems, which are necessary for packing and unpacking complex data types.

5.3 Performance and Other Myths of Layered Software

Performance. In many domains, users are willing to trade a certain amount of performance for the benefits of a software engineering environment. In the networking domain and others, this is emphatically not the case.

Network protocol design and implementation is performance-driven to the extreme. The design of Avoca reflects this fact. Avoca, far from being a top down exercise in software design, is the side-effect of a performance experiment.

It was a widely-held belief in the network community that encapsulated and layered protocol design inflicts a prohibitive performance penalty [61]. The Avoca domain model was a side effect of an experiment designed to show that this belief was untrue or overstated. It involved the decomposition of the monolithic Sprite RPC protocol into thirty or so microprotocols; a model was required simply to manage the large number of protocols. The majority of the work on Avoca was focused on making highly layered protocol suites efficient. While much of this work is protocol-specific (and in any case, there is too much of it to be presented here—see papers on the *x*kernel), some of it is applicable to the general problem of fast layered systems.

The first lesson is that the temptation to make each layer a process has to be avoided at all costs. When organized in this fashion, traversing each additional layer costs at least one context switch. Context switches even for 'lightweight' processes are enormously expensive when one considers the effect on a memory cache [44]. Most interprocess communication primitives have additional built-in overheads that only make matters worse. The *x*kernel uses a process-per-message as it enters the kernel. This process shepherds the message through a graph of protocol objects until it leaves the kernel. In general, only one context switch is required for each message regardless of the number of layers.

The second lesson is the cost of copying data and the power of shared abstractions. Memory-to-memory copies of large data blocks (500 bytes) are expensive. Any layered protocol design that requires data to be copied from layer to layer would be prohibitively slow. The *x*kernel message abstraction (i.e., class/ADT) manages message data buffers in a lazy fashion by avoiding copies whenever possible and sometimes by eliminating copying altogether. All Avoca protocols use a common message class/ADT to manipulate messages.

Finally, while shared classes/ADTs are often looked upon as time-savers that avoid duplicate work, we found that sharing classes is good not because it reduces the total amount of work, but because when a class is constantly being reused, any performance or correctness problems will be found and fixed. Consequently, such fixes improve the reliability and performance of all components that use the class, thereby greatly increasing the potential performance gain over component specific improvements.

The approach to developing Avoca was to produce the fastest possible protocol suites, and against all preconceived notions (including the designers), the decomposed version of Sprite RPC turned out to be faster than the original and monolithic version in both latency and bandwidth (see [45, 46] for actual performance figures). This is not to say that protocol encapsulation improves performance in most cases, but that the per protocol overhead was low enough that it did not dominate the performance outcome. *Avoca demonstrates that careful attention to performance can lead to component-based* *systems whose performance is equal or even superior to monolithic ad-hoc systems.*

Decomposition. A classical fallacy that has been ingrained in the computer science culture is the belief that hierarchical decomposition of software is impossible if performance-competitive algorithms are used. First, decomposition of software into "conceptual" or "virtual" layers is *always* possible. Decomposition is a *correctness* issue, not a performance issue. The means by which one decomposes an inefficient software system is identical to those used for decomposing an efficient system; hierarchical decomposition has nothing to do with performance.

Second, it happens all the time that components will have various degrees of composability. Some components have little or no constraints on their use. Other components can only work in the presence (or *cannot* work in the presence) of a specific class of components. All of these possibilities arise in practice. This is the reason why design rule checking is present in the Genesis layout editor—these rules preclude incorrect combinations of components from being specified. We will say more about design rule checking shortly. But again, the fact that dependencies between components exist does not preclude a "hierarchical" system decomposition.

5.4 Other Issues

Encapsulation. Our work distinguishes two types of encapsulation: ADT and design. We are all familiar with *ADT encapsulation*; programming languages (ADA, C ++) enforce ADT encapsulation through the hiding of local variables and local functions [41]. The idea is to permit changes to an ADT's implementation without triggering changes to other ADTs. *Design encapsulation*, on the other hand, is the form of encapsulation that is required for software component technologies. It is the hiding of design details within components and, as we will see below, is the result of a deliberate design. Unlike ADT encapsulation, there are aspects of design encapsulation that are *not* enforceable by compilers. Consider the following examples.

Two commonly identified ADTs (modules) in a DBMS are the storage system (which stores and retrieves tuples of relations) and the query optimizer (which identifies the fastest search strategy to be used by the storage system in processing a query) [39]. Both the storage system and query optimizer can be implemented by distinct ADTs, where each other's local variables and local functions are hidden from view. Although ADT encapsulation is preserved, design encapsulation is not. In general, the query optimizer needs to know *exactly* what structures are maintained by the storage system in order to estimate retrieval costs. Adding or deleting a storage structure must always be accompanied by a corresponding change to the query optimizer because design information is shared by both. Thus, design encapsulation transcends the encapsulation of individual ADTs/classes, and normally deals with sets of closely related ADTs. We cannot imagine how design encapsulation in this example could be enforced by compilers.

As a second example, the data definition language (DDL) of a DBMS is used to declare schemas and hints to the DBMS on how to store and retrieve data efficiently. For example, if an indexing component is present in a DBMS, there will be a special statement in the DDL allowing schema designers to declare that specific fields are to be indexed. If the indexing component is absent, the indexing statement is not present in the DDL.

A standard way to implement DDLs is via Lex and Yacc [43]. An individual DDL statement is represented by one or more Lex tokens and one or more Yacc rules. The presence or absence of a component will trigger the inclusion or exclusion of tokens and rules. The inclusion/exclusion process must be done at precompile time, a task that has nothing to do with ADT encapsulation. Thus, design encapsulation cannot always be realized solely by ADT encapsulation.

We believe that Parnas was the first to recognize the concept of design encapsulation [50], and Guttag was among the first to recognize the concept of ADT encapsulation [31]. However, the distinction between design and ADT encapsulation was not evident to us before we began the Genesis and Avoca projects.

Domain modeling. Creating a domain model is definitely not an ad hoc process. It involves a careful study of existing systems within a domain to discern the fundamental layering present in all such systems and a disentanglement of abstruse designs of ad hoc systems into a standard language (i.e., type expressions). Using existing systems as a guide to standardize the decomposition of systems and designing generic interfaces for components/realms is the essence of domain modeling.

There is no standard notation or metamodel for expressing a domain model. For the few domain models that exist, each is expressed in terms of its own special-purpose concepts, which not only complicates any modeling effort (i.e., identification of an appropriate notation is nontrivial), but it also makes model comprehension difficult (because ad hoc notations and metamodels are rarely similar).

Our type-expression notation and our component-realm metamodel are domain independent. We have shown that a common set of ideas apply to network software and database software, even though both domains are vastly different. We conjecture that our notation and metamodel are applicable to a large class of domains (i.e., those with hierarchical systems), and they present a concrete language for domain analysts to express the results of their domain models.

Design rule checking. Not all combinations of components are meaningful. Indiscriminate but syntactically correct type expressions may correspond to systems that cannot possibly work. An integral part of software component technologies are layout editors that enforce *design rules* that preclude illegal combinations of components. The basic idea is simple enough: every component c may insist that lower-level components (which instantiate c's parameters) satisfy constraints *in addition to* that of type membership. That is, if c[x: R] has parameter x of type R, it may be that not all components of R can legally instantiate x; only those with certain properties are permitted. Design rules express these constraints and confirm common intuition: as one composes components, constraints on legal instantiations are 'inherited' in a top-down manner, thus further restricting the set of components that can be used at any given position in a system.

We explain in [9] how design rules are defined and enforced in the Genesis component layout editor (DaTE). DaTE presents a graphical interface by which system designs can compose components quickly, while ensuring that incorrect compositions are impossible to specify. As is evident from Section 2, type expressions for typical systems are difficult to read. Graphically visualizing expressions significantly simplifies understanding and interpretation of type expressions, and reinforces the 'lego' paradigm that underlies software components.

Since our type-expression formalism is generic, it is possible that domain-independent layout editors can be built. Such editors will be table driven, where tables encode domain-specific knowledge. As more domains are reduced to a type-expression formalism, we expect to better understand the features that these editors will need to provide.

Component interconnection. Component interconnections can be realized in a variety of efficient ways. Both Avoca and Genesis use dispatch tables that encode the static topology of stacked components. Thus, connections between components are determined at runtime. This implementation strategy easily supports systems where a component is referenced several times (e.g., m[a, a]). Only one copy of the component's source code will be present, but multiple execution instances will exist. Control blocks are used to differentiate multiple execution instances.

Whenever there is a choice of different downward paths to follow, as occurs in Avoca virtual protocols and Genesis software busses, static dispatch tables are not sufficient and additional connectivity information must be stored. Such information is recorded in component-internal tables and pointers in control blocks. Avoca goes further by using additional dynamically-defined tables (called Maps) to support upward paths arising from upcalls.

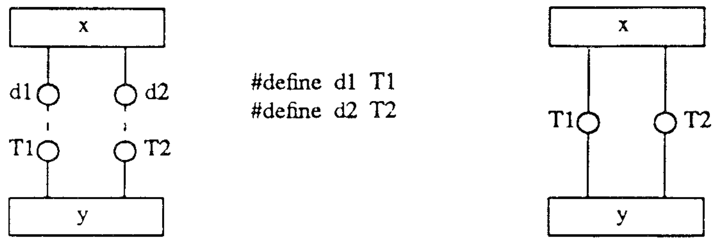
When possible, Genesis also fuses components together at compile-time by a very simple technique. In the composition x[y], component x calls y. This means that each downcall of d_i of x is a call to the top operation T_i of y. Creating a set of C precompiler statements, one for each downcall of x, of the form:

```
#define d_i T_i
```

will literally rename each d_i in x with T_i, thereby making x call y directly. Figure 10 illustrates component fusion.

6. CONCLUSIONS

The success of mature engineering disciplines lies in the standardization of well-understood technologies. By purchasing off-the-shelf components, engineers can create customized systems economically by building only the parts



```
#define d1 T1
#define d2 T2
```

(a) before fusion (b) after fusion

Fig. 10. Component Fusion via Call Renaming.

that are application-specific. Unnecessary reinvention of technology is avoided.

Software engineering (and computer science itself) is a relatively young and immature discipline. Contemporary software systems have been simple enough that massive technology reinvention was economically feasible. This will not be true of the future. There are many domains today that are technologically mature and ripe for standardization, and there will certainly be many more a decade from now. Many of these domains will center on hierarchical systems, where a progression of increasingly more sophisticated software technologies are layered upon each other.

We have presented a *validated* model of hierarchical system design and implementation that explains how families of such systems can be assembled quickly and economically using prefabricated components (i.e., software ICs, software legos). Our model is aimed specifically at mature software technologies, where standardization makes sense. Standardization is the key to large scale reuse.

We have shown that complex domains can be expressed by an elementary metamodel of realms of interchangeable and plug-compatible components. Software systems of enormous complexity are elegantly represented as type expressions. We have demonstrated that symmetric components are the key to large-scale reuse in important domains (e.g., unix file filters, database FMAP components, communication protocols). These components have the unusual property that they can be composed in virtually arbitrary orders.

We have noted common features that all software component technologies should exhibit, and have identified three distinct granularities of 'as-is' reuse: component, class, and algorithm. We explained how our concepts reinforce and extend the pioneering ideas of Parnas and Habermann et al. in the design of hierarchical systems, and how these ideas transcend current object-oriented design techniques. Specifically, we used the ER model and notation as a leverage to present additional concepts that are needed to elevate the fundamental unit of large scale software construction from individual classes to parametric components of highly interrelated classes.

We explained that highly layered systems can be as efficient as their monolithic counterparts and that software components require design encapsulation—a form of encapsulation that is different than conventional ADT encapsulation. Because our type-expression notation and component-realm metamodel are domain independent, we also feel that our work can serve as the basis for a common language in which other domain models can be expressed.

We conjecture that our model is a blue-print for a practical software components technology. The first major test of this conjecture will be the DARPA-funded upgrade of the Mach operating system that will replace Mach's current communication kernel with that of Avoca/*x*kernel. This upgrade, which is planned to be the standard release for a future version of Mach OS, is now underway at the University of Arizona.

ACKNOWLEDGMENTS

We are grateful to Bob Schwanke, Don Fussell, Ira Baxter, Paul Clements and the referees for their suggestions on improving earlier drafts of this paper. We also thank Jay Roy, Rob Read, Ham Richards, and Kelsey Shepherd for their helpful comments.

REFERENCES

1. BACH, M. J. *The Design of the Unix Operating System* Prentice-Hall, Englewood Cliffs, NJ.,1986.
2. BATORY, D. S. Modeling the storage architectures of commercial database systems. *ACM Trans. Database Sys. 10*, 4 (Dec 1985), 463–528.
3. BATORY, D. S. Extensible cost models and query optimization in GENESIS. *IEEE Database Eng.* (1987).
4. BATORY, D. S. Concepts for a database system compiler. *ACM PODS*, 1988.
5. BATORY, D. S., BARNETT, J R., GARZA, J. F., SMITH, K. P., TSUKUDA, K., TWICHELL, B. C , AND WISE, T. E. GENESIS: An extensible database management system *IEEE Trans. Softw. Eng.* (1988), 1711–1730.
6. BATORY, D. S., BARNETT, J. R., ROY, J., TWICHELL, B. C., AND GARZA, J. Construction of file management systems from software components. *COMPSAC*, 1989
7. BATORY, D. S. On the reusability of query optimization algorithms. *Inf. Syst.* (1989).
8. BATORY, D. S. The genesis database system compiler: User manual. Univ. of Texas Tech. Rep. TR-90-27, 1990.
9. BATORY, D. S., AND BARNETT, J. R. DaTE. The genesis DBMS software layout editor. In *Conceptual Modelling, Databases, and CASE*, R. Zicari, Ed., McGraw-Hill, 1991.
10. BATORY, D. S. A domain modeling methodology. In preparation.
11. BATORY, D. S., ET AL. The implementation of genesis. In preparation
12. BAXTER, I. Personal communication.
13. BERNSTEIN, P. A., AND CHIU, D. M. Using semi-joins to solve relational queries. *J. ACM. 28*, 1 (Jan. 1981).
14. BERNSTEIN, P. A., ET AL. Query processing in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 6*, 4 (Dec. 1981).
15. BERSHAD, B., ANDERSON, T., LAZOWSKA, E., AND LEVY, H. Lightweight remote procedure call. In *ACM Symposium on Operating Systems Principles* (Dec. 1989), 102–113.
16. BIGGERSTAFF, T. J., AND PERLIS, A. J. *Software Reusability I: Concepts and Models.* ACM Press, 1989.
17. BIGGERSTAFF, T. J., AND PERLIS, A. J. *Software Reusability II Applications and Experience.* ACM Press, 1989.
18. BOOCH, G. *Software Components with Ada.* Benjamin Cummings, 1987.
19. BOOCH, G. *Object-Oriented Design With Applications.* Benjamin Cummings, 1991.

20. BRITTON, K. H., PARKER, R. A., AND PARNAS, D. L. A procedure for designing abstract interfaces for device interface modules. In *International Conference on Software Engineering* (1981) 195–204.
21. CARDELLI, L., AND WEGNER, P. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv. 17*, 4 (Dec. 1985).
22. CHAMBERLIN, D. D., ET AL. SEQUEL 2: A unified approach to data definition, manipulation, and control. *IBM J. Res. Dev. 20*, 6 (Nov. 1976).
23. CHERITON, D. R. VMTP: A transport protocol for the next generation of communications systems. In *ACM SIGCOM '86 Symposium* (Aug. 1987), 406–415.
24. CLARK, D. D. The structuring of systems using upcalls. In *ACM Symposium on Operating Systems Principles* (Dec. 1985), 171–180.
25. DEUTSCH, P. L. Design reuse and frameworks in the smalltalk-80 programming system. In *Software Reusability I: Concepts and Models*, ACM Press, 1989.
26. DIJKSTRA, E. W. The structure of THE multiprogramming system. *Commun. ACM 11*, 5 (May 1968).
27. GOGUEN, J. Parameterized programming. *IEEE Trans. Softw. Eng. SE-10*, 5 (Sept. 1984).
28. GOLDBERG, A. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984.
29. GRAEFE, G., AND DEWITT, D. J. The EXODUS optimizer generator. *ACM SIGMOD*, 1987.
30. GRAVES, H., AND POLAK, W. Common intermediate design language overview. Lockheed Palo Alto Res. Lab., Spring 1991.
31. GUTTAG, J. Abstract data types and the development of data structures. *Commun. ACM* 20, 6 (June 1977).
32. GUTTAG, J., HOROWITZ, E., AND MUSSER, D. R. Abstract data types and software validation. *Commun. ACM* 21, 12 (Dec. 1978).
33. HABERMANN, A. N., FLON, L., AND COOPRIDER, L. Modularization and hierarchy in a family of operating systems. *Commun. ACM 19*, 5 (May 1976).
34. HABERMANN, A. N., FLON, L., AND COOPRIDER, L. Modularization and hierarchy in a family of operating systems. *Commun. ACM 19*, 5 (May 1976).
35. HAYES, R., MANWEILER, S., AND SCHLICHTING, R. A simple system for constructing distributed, mixed language programs. *Softw.—Pract. Exper.* (July 1988).
36. HUTCHINSON, N. C., AND PETERSON, L. L. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Trans. Softw. Eng. 17*, 1 (Jan. 1991).
37. JARKE, M., AND KOCH, J. Query optimization in database systems. *ACM Comput. Surv. 16*, 2 (June 1984).
38. JOHNSON, R. E., AND FOOTE, B. Designing reusable classes. *J. Object-Oriented Program.* (June/July 1988).
39. KORTH, H. F., AND SILBERSCHATZ, A. *Database System Concepts.* McGraw-Hill, 1991.
40. LINDSAY, B. Private communication, 1989.
41. LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (Aug. 1977).
42. LOHMAN, G. Grammar-like functional rules for representing query optimization alternatives, In *Proceedings of ACM SIGMOD 1988* (Chicago, June 1988), 18–27.
43. MASON, T., AND BROWN, D. *Unix Programming Tools: Lex & Yacc.* O'Reilly and Associates, 1990.
44. MOGUL, J. C., AND BORG, A. The effect of context switches on cache performance. In *Conference on Architecdtural Support for Programming Languages and Operating Systems* (April 1991), 39–51.
45. O'MALLEY, S. W. Ph.D. dissertation, Univ. of Arizona, 1990.
46. O'MALLEY, S. W. Avoca: An environment for programming with protocols. Ph.D. dissertation, Univ. of Arizona, TR90-31, Aug. 1990.
47. O'MALLEY, S. W., AND PETERSON, L. L. A new methodology for designing network software. Submitted for publication.
48. O'MALLEY, S. W., ABBOTT, M. B., HUTCHINSON, N. C., AND PETERSON, L. L. A transparent blast facility. *J. Internetworking, 1*, 2 (Dec. 1990).

49. OUSTERHOUT, J. K., CHERENSON, A. R., DOUGLIS, F., NELSON, M. N., AND WELCH, B. B. The Sprite network operating system. *IEEE Computer*, (Feb. 1988).

50. PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15*, 12 (Dec. 1972).

51. PARNAS, D. L. Designing software for ease of extension and contraction. *IEEE Trans. Softw. Eng.* (Mar. 1979).

52. PRIETO-DIAZ, R., AND ARANGO, G., ED. *Domain Analysis and Software Systems Modeling.* IEEE Computer Society Press, 1991.

53 PURTILO, J. Polylith: An environment to support management of tool interfaces, In *Proceedings of the ACM SIGPLAN Symposium on Language Issues in Programming Environments* (July 1985), 12–18.

54. ROY, J. Design and use of the Jupiter file management system. M.Sc. thesis, Dept. of Computer Science, Univ. of Texas, 1991.

55. RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

56. SOFTWARE ENGINEERING INSTITUTE. *Proceedings of the Workshop on Domain-Specific Software Architectures* (July 9–12, 1990).

57. SELINGER, P. G., ET AL. Access path selection in a relational database management system. *ACM SIGMOD,* 1979.

58. SNODGRASS, R. *The Interface Description Language: Definition and Use.* Computer Science Press, Rockville, Md., 1989.

59. STONEBRAKER, M., WONG, E., KREPS, P., AND HELD, G. The design and implementation of INGRES. *ACM Trans. Database Syst 1*, 3 (Sept 1976)

60. TANENBAUM, A. S. *Computer Networks.* Prentice-Hall, 1988.

61. TENNENHOUSE, D. L. Layered multiplexing considered harmful. In *1st International Workshop on High-Speed Networks* (Nov 1989).

62. TEOREY, T. J., YANG, D., AND FRY, J. P. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Comput. Surv. 18*, 2 (June 1986).

63. TEOREY, T. J. *Database Modeling and Design: The Entity-Relationship Approach.* Morgan-Kaufmann, 1990

64. TSICHRITZIS, D., AND KLUG, A., EDS. *The ANSI / X3 / SPARC DBMS Framework.* AFIPS Press, 1978.

65. WADLER, P., AND BLOTT, S. How to make ad-hoc polymorphism less ad hoc. *ACM POPL,* 1989.

66. WONG, E., AND YOUSEFFI, K. Decomposition—A strategy for query processing. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976).

67. YU, C. T., OZSOYOGLU, Z. M., AND LAM, K. Optimization of tree queries. *J. Comput. Syst. Sci. 29*, 3 (Dec. 1984).

68. ZANIOLO, C. The Database Language GEM. *ACM SIGMOD*, 1983

69. ZIMMERMANN, H. OSI Reference Model—The ISO model of architecture for open systems interconnection. *IEEE Trans. Commun. 28*, 4 (April 1980).

Received June 1991; revised January and May 1992; accepted June 1992