Foundation for a C++ Programming Environment

Richard P. Gabriel
Nickieben Bourbaki
Matthieu Devin
Patrick Dussud
David N. Gray
Harlan B. Sexton

Lucid, Inc.

Abstract

Over the last two years we have been designing and implementing an architecture for environments along with an instantiation of that architecture as a C++ programming environment. To do this we have examined the most effective environments—programming or otherwise—and gathered features we would like to see in next-generation environments, and from this we have designed a minimal set of constructs. Among the most difficult to achieve is tight integration without implementing all tools as a monolith.

The result is a prototype of a very tightly integrated C++ environment to which it is easy to connect additional C++ programming tools, CASE tools, configuration management tools, and documentation support tools. The environment can be easily extended for multi-programmer groups.

Written entirely in C++, this environment uses advanced object-oriented programming techniques and provides persistent C++ instances. A new paradigm for integration—*annotations*—has been developed which unifies control integration, data integration, and user interfaces.

1. Introduction

Over the last two years we have been designing and implementing an architecture for environments along with an instantiation of that architecture as a C++ programming environment. To do this we have examined the most effective environments—programming or otherwise—and gathered features we would like to see in next-generation environments, and from this we have designed a minimal set of constructs. Among the most difficult to achieve is tight integration without implementing all tools as a monolith.

Our project is named “Cadillac”, though we expect that this name will be dropped if we make a product. Because this paper is about programming environments, we will use the term “Cadillac” to refer to the programming environment, and we will point out specifically when we are referring to a generic architecture.

1.1 *Brief Overview*

An environment is a set of tools integrated harmoniously for the purpose of accomplishing some task or set of tasks—in particular, this set of tools must be able to work well together as judged by a user of the environment. Often, this is simply being able to share information among tools or to control the activation of them, yet this overlooks problems of software lifecycle and effective presentation of information.

The Cadillac programming environment is a set of tools designed to facilitate the tasks associated with developing software. These tools include, but are not limited to, editors, compilers, debuggers, and linkers. Even more important than the tools provided by Cadillac is the mechanism by which these tools are integrated. This mechanism provides a well-defined, flexible, and extensible architecture for the components of the environment.

Coding is at most only 10% of the software lifecycle, but maintaining and modifying that code is a larger fraction, and one that persists over long periods. In the DoD, some software must be maintained and altered over a 25-year period. So an environment for initial development might not significantly reduce effort over the entire lifecycle. Though we do not have specific tools that directly solve this problem, we provide mechanisms that will support such tools by persistently retaining information about the software, and these mechanisms are helpful even without fancy layered tools.

The current trend in environments can be characterized (or caricatured) as “let a thousand windows spawn.” Some environments create a new window for every command. The user is required to maintain these windows on a messy desk. One of our goals has been to design constructs that group information so that presentation can be concise and uncluttered. Maintaining the screen should not occupy the user.

In this paper we describe the architecture and its mechanisms, and we also provide comparison with other work. A prototype exists of this environment, and we are just now taking the significant step of using the environment on its own evolving implementation.

1.2 *Tight versus Loose coupling*

Clearly, the requirements of such an architecture are driven by both the needs and expectations of those using the system and those extending it. Specifically, for the user a programming environment must provide tight integration of tools and the information they provide. One way to accomplish this is with a *closed implementation* of the environment in which the programming tools are designed together to work on the same representation. A closed system is one that is constructed like a black box—no internal parts are visible, alterable, customizable, and no external parts can be added without source access.

For example, in a closed programming environment a *procedure* might be represented as a structure containing source code, a parse tree for the code, a symbol table, and possibly some other parts. When the editor makes changes to the source code, the compiler could make changes to the parse tree, updating the symbol table. Here the tools are tightly intertwined. Though succeeding for users under most circumstances, this approach fails those who would augment the system, and it generally proves to be more difficult to make such tightly coupled environments work well on distributed computer systems.

In the Cadillac environment tools are clients served by a kernel that effects the integration. Tools and the kernel do not operate on the same representation. Instead they operate on the same abstract interface to shared information. In other words, the kernel handles keeping track of the information derived from the tools and relationships among such information, and the (client) tools provide the information and the relationships.

1.3 *Rationale*

The approach chosen by Cadillac is a combination of *control integration* and *data integration*. Control integration effects collaboration of distinct tools, especially in a distributed computing environment. Data integration effects retaining and sharing of data relationships and program state by distinct tools over time. Rather than viewing these two approaches as complementary, we feel they are connected, and we have developed an integration mechanism that merges them.

In control-integration-based environments such as Field [Reiss 1989] and SoftBench [Cagan 1989] [Gerety 1989], the environment acts as a message-routing network, routing input to and output from the various tools (and users), while leaving it up to the individual tools to coordinate information about state that is too big or complex to be passed in a message. (For example, in SoftBench [SB1], the static analyzer, compiler, and debugger share information about program structure generated by the compiler—this information is not intrinsic to the SoftBench system in the same way that message protocols are.) In Cadillac, all integration information is maintained in the kernel (via a persistent object system), so that each tool needing this information has access to it.

There are three reasons for trying to combine control and data integration. First, process control mechanisms have proven remarkably successful in the Field and SoftBench systems, providing good integration between disparate, pre-existing tools while being extensible. On the other hand, many tools such as program analyzers, incremental compilers, and documentation systems, require substantial shared state to work effectively, and control-only integration provides little help with these data-dependent tools.

The second reason for providing a data/control mechanism is that if one has a mechanism that combines control and data—that is, structure and behavior—one has the basis for a user interface mechanism. Because this mechanism is used for integration, its use as a user interface mechanism implies a concise, consistent environment. In fact, these three facets—control integration, data integration, and user interface integration—are combined in a mechanism we call *annotations*[1]. In brief, an annotation is an object that links objects. One can think of annotations as generalized hypertext.

The third reason is that some tools can be implemented only in terms of the 'glue' between external tools. For example, navigation—moving between related functions, classes, declarations—is provided by a tool that operates on the gathered output of a code analyzer and interacts with the user by using a presentation system. The code analyzer might be a compiler, and the presentation system a grapher, as is the case with Cadillac, and it is unreasonable to expect that either the compiler or the grapher should be modified to be a navigator.

2. Architecture

The Cadillac environment is a foundation or framework for environments. Refer to Figure 1. The environment consists of a *kernel* that provides integration and persistence and a set of tools, called *clients*, that communicate with the kernel through a set of tool-class-specific protocols. The kernel keeps track of the information derived from the tools and relationships among such information. The clients provide the semantic underpinning of the environment.

[1] Our notion of annotation is different from the one used in Field.

Each client has minimal knowledge about the kernel. Similarly, the kernel has minimal knowledge about its clients. The knowledge of the kernel is in the form of a set of protocols that define each tool. These protocols can be viewed as an abstract data type (ADT), where the actual tool corresponds to its implementation.

Each protocol is designed to capture the essence of the activities of a particular kind of tool, and as far as is possible, the expertise regarding a tool's domain remain with that tool.

For example, a good programming environment has knowledge about the syntax and semantics of the programming language it handles. Rather than build this knowledge into the kernel, the kernel is programmed to know about an abstraction on programming languages, which is a set of what we call *language elements*: Toplevel forms, function definitions, class definitions, declarations, interfaces, implementations, function invocations, and the like are language elements. The kernel submits source text to the compiler by using the *compiler protocol*, and the compiler returns the textual positions of the language elements it finds. By using annotations the relationships between the source text and the language elements are maintained. The effect is that the environment knows about the program and the language it is in, but this knowledge is gained from the compiler and from nowhere else. In later sections we will look at the details of protocols and annotations.

Any compiler that follows the compiler protocol can be used as the compiler for a Cadillac environment, even if that language is not C++. In ADT terminology, the kernel does not care about the implementation of the compiler.

As much as possible a tool protocol is designed to model the normal behavior and operation of the tool it represents.

Other clients in a programming environment besides the compiler are its associated debugging, instrumentation, and inspection tools. The presentation client is responsible for presenting information, providing navigational aids for that information, and providing mechanisms to edit or alter the program source or its associated information.

3. Annotations

The key to integration in Cadillac is the use of *annotations.* Annotations are a generalization of hypertext. An annotation is an object along with an associated set or sequence of other objects. An annotation acts as a link among those other objects. Each annotation is an instance of a class to which methods can be attached. An example annotation is the simple link which is associated with a sequence of two objects. The first of the two objects is the *source* and the other is the *destination* of the link. When both objects are text, this is merely hypertext.

More formally, an annotation is a pair (a, s), where a is a member of the class[2] $\mathcal{A}$, which represents annotations, and s is either a set $\{o_1, \ldots, o_n\}$ or a sequence $\langle o_1, \ldots, o_n \rangle$, where each o_i is an instance of some class. Furthermore, $\exists\, f\colon \mathcal{A} \to \mathcal{S}$, where $\mathcal{S}$ is the set of all subsets and subsequences of objects, such that if (a, s) is an annotation, $f(a) = s$. Therefore, we can unambiguously denote the annotation (a, s) by a.

Annotations can link subobjects as well, but this can be formalized by creating objects to represent the subobjects. An example of this is an annotation that links two pieces of text, and the corresponding subobjects are called *extents*. An extent is a contiguous region of text to which annotations can be attached. Extents are allowed to arbitrarily overlap, and several annotations can be attached to the same extent.

Furthermore, there is a function `an` such that given any object o, $\texttt{an}(o) = \{(a, s) \mid a \in \mathcal{A},\ o \in s\}$. That is, given any object it is possible to find all annotations involving it.

Annotations are first-class objects in the kernel, and can be themselves annotated. New annotation classes can be created and methods written for them (though the current prototype does not have an extension language for this).

We have not yet determined the advantages and disadvantages of providing fully indexed annotations, with which, for example, it would be possible to find all annotations involving several given objects.

The class of an annotation provides a set of operations that can be performed on instances of that class. The operations on an annotation are carried out by the kernel. The results of carrying out an operation often result in the presentation client presenting new or altered material to the user.

[2] A class can be taken as the set of its members

3.1 *Annotations are Active*

An annotation is *active*: Methods are attached to annotations by being associated with a class. When a tool, such as a compiler, is being sent source text, any annotation encountered during the process may have an associated method invoked to determine the text to be sent in its place. For example, region locking could be implemented with annotation methods. When locked text is sent to the presentation client, the method associated with the annotation causes the correct commands to be sent to the presentation client to cause the text to be locked in the editor. In order to implement such functionality, the presentation client and other tools would need to preserve the semantics associated with region locking.

Typically, methods are run in the kernel, possibly in the tool protocol servers, but such protocol servers can cause methods to run in some client.

3.2 *Abbreviations*

Annotations can be abbreviated. An *abbreviation* can be thought of as an alternative means of display. When an annotated object is displayed (by the *projection* operator to be described in the kernel section), the presentation depends on the object and its annotations.

For example, comments sometimes take up too much of the screen. If there were a language element for comments, then the display of a comment could depend on that language element. If comments are to be elided, the comment language element could be annotated to reflect this and the abbreviation mechanism could be used to elide comments.

Here, an abbreviation is a means either to completely hide a comment or to indicate it with a simple icon. From the icon the original comment text should be available, and this implies that when a comment is projected, a new object is created that represents the abbreviation, and that new object inherits the annotations of the original object and is additionally annotated with the original object.

Let $\mathcal{B}$ be the class of abbreviations, `project`: $\mathcal{O} \times \mathcal{T} \to \mathcal{D}$ be the projection operator that takes an instance and a tool domain and projects that object vis-à-vis the tool's domain onto the tool projection domain, `inherit`: $\mathcal{O} \times \mathcal{O} \to \mathcal{O}$ be the annotation inheritance operator which takes two objects and causes the first argument to inherit the annotations of the second argument (returning its first argument), and `annotate`: $\mathcal{A} \times \mathcal{O} \times \mathcal{O} \to \mathcal{A}$ be the binary annotation function that takes a class of annotation and two objects and annotates the first object with the second by an annotation of the specified class. Then presenting an object o that should be abbreviated is specified as follows:

$$\text{let}\, n \in \mathcal{B};\ \texttt{project}(\texttt{annotate}(\mathcal{B}, \texttt{inherit}(n, o), o), t)$$

A new instance n of $\mathcal{B}$ is created to represent o, the annotations of o are added to n, n is annotated with o as an abbreviation, and n is projected onto some t in $\mathcal{T}$.

Abbreviations can be more elaborate. The material included in a presentation of an object depends on the type of the annotation, the type of the tool doing the presentation, and the types of the objects linked by the annotation. This way, there may be a number of different manifestations or views of the same linked source.

A good example is fully instrumented programs. Sometimes a program can be used for several purposes. For example, a simulation program might be used to determine the final state after a complex series of actions and interactions. That program might also be used to determine intermediate states or to produce intermediate statistics, or even to produce statistics about its own behavior. These secondary uses could be implemented by annotations on the original code. These annotations would be in the form of alternative code fragments sprinkled through the original code. When being used for its original purpose, the original code can be used, and when being used for a secondary purpose, the code with the alternative code fragments substituted for the originals could be used.

4. Kernel

The kernel consists of three basic parts, a part that persistently stores objects that represent source text, language elements, compiled code objects, and annotations, a part that consists of the set of tool protocol interfaces, and a part comprised of high level tools, such as the class browser.

Some of the represented objects, such as source text, are stored in the native file system, and the objects that represent them can be viewed as surrogates. There are two implementations of the persistent object store, one based on an ISAM (Sun NetISAM and C-ISAM) and the other based on a commercial object-oriented database (OODB).[3]

[3] The ISAM version was initially written because no commercial OODB was then capable of supporting Cadillac. The ISAM version is discussed in a later section, and it is worth noting that the recent port to the OODB took only a few days.

Later versions of Cadillac will use a multi-user database for group projects.

Each tool client is served by a protocol server or adapter that communicates with the tool. Communication is through a socket, so that a tool can be running on a different computer from the kernel. The tool can be written in any language because only the socket-based communication protocol is important to the kernel.[4]

Associated with each tool is a set of native types: a native type is a kernel class, and the set of native types for a tool is the *natural domain* of that tool. For example, the compiler tool operates on language elements and source text. Instances of native types are stored in the database and usually defined as subclasses of very general kernel classes. (Recall that source text resides in the file system and that surrogate objects populate the kernel.)

The set of native types is the distillation of what the various tools need to know in common about a program, so it is necessary for some of these tools to be able to operate on the natural domains of other tools. Projections accomplish this: a projection presents an object to a non-native tool so that it can be manipulated by that tool.

For example, an interactive tool presents a view of a native object to the user, proposes choices, and acts on user choices to alter this object. A projection method is responsible for transforming (called *projecting*) a non-native object into a native one, and then altering the non-native object in response to changes that occur to the projected object (called *unprojecting*). The interactive tool has only to present the projection to the user and manage the user interface. The kernel maintains dependency links between objects, their projections, and tools acting on the projections, in order to keep all the views on an object synchronized with the object—this *dependency registration* protocol is similar to that used in Smalltalk environments.

Strictly speaking there are two types of projection: the primary projection projects a native type of one tool to the native type of another tool, and the secondary projection projects the native type of a tool to the tool itself. This latter projection is via the tool protocol and can thought of as an isomorphism, and when we talk about projections we will mean primary projections.

[4] The communication protocol currently depends on the byte order being the same on communicating machines.

The projection operator and its inverse (which is itself) are *generic* in the sense of the Common Lisp Object System (CLOS)—that is, these operators are functions of two arguments, both of which participate in method selection. There is a discriminator function `discriminate`: $\mathcal{C} \times \mathcal{C} \rightarrow \mathcal{M}$ which takes two classes and returns the applicable method. Each object that is an instance of a native type has a field that contains its class, so that there is a function that could be called `class-of` that takes an instance and returns its class. Therefore projection is defined as follows:

$$\texttt{project}(o, d) := \texttt{discriminate}(\texttt{class-of}(o), \texttt{class-of}(d))(o, d)$$

where o is the object to be projected and d is the domain to which it will be projected. The function actually applied is computed by the discriminator. The generic function `discriminate` is implemented by hash tables. The entry for the first class in the first hash table is a second hash table that maps the second class to the appropriate method.

In this way, projections are inherited.

The kernel is implemented in C++, and all instances (annotations, native types, language elements, etc) are C++ instances.

5. Tool Servers

The standard client tools include a text editor, a grapher, a class browser, a compiler, a debugger, a dynamic/incremental loader, and an on-the-fly Unix manual page annotator[5]. Any tool that obeys a particular protocol can be used as a client of that type. New protocols can be dynamically loaded into the kernel.

There is not currently an extension language in which new protocol servers can be written, though such a facility will be added.

[5] This facility looks up library functions in the Unix manual (by using the `man` command), and parses the output into kernel objects. These objects are displayed with multiple fonts in the text editor along with automatically generated annotations. For example, references to other manual pages are annotated with a lazy annotation that connects the reference to the referenced page. References to library source files are annotated with the functions within those files. This tool is written in 300 lines of C++.

6. Persistent Object System

The Persistent Object System (POS) is a facility for making C++ objects *persistent*. That is, persistent C++ objects exist longer than a single invocation of Cadillac, and this is implemented by using a database to store such instances. This facility enables the environment to retain information, such as dependency analyses, over long periods of time, for example, the lifetime of the software.

The POS is a layered program, with a small number of layers with different responsibilities. The layers, from highest to lowest level, are as follows:

- Application Programming Layer
- Persistent Object Class Definition Layer
- Database Interface Layer
- Database Implementation Layer

The highest layer is the application programming layer. At this level the programmer merely has to use object types and corresponding pointer types defined at lower levels. The application programmer need never be concerned about whether objects are or are not in memory, or about explicitly accessing the database.

The next lower layer is the persistent object class definition layer. At this level new object classes are defined to the POS, and methods are defined for loading and storing objects in their corresponding database file or files. These methods are defined in terms of the functionality provided by lower layers.

The database interface layer provides the next higher level with a portable functional interface to the underlying database system. This interface is primarily useful in implementing loading and storing methods.

Finally, the lowest level is the database implementation layer. This is the underlying database system. There are minimal functional requirements on this system, including the requirement that simple accesses based on single integer keys be relatively fast.

By defining new classes of objects that inherit from persistent classes defined in the POS, the programmer can create *smart objects*, and associated classes of *smart pointers*, which have properties of persistence, resource-allocation, reference-counting garbage collection, and a least recently used (LRU) object-swapping capability. The fundamental implementation technique is to distribute the work between the smart pointer classes and the persistent object classes to which such pointers point.

The operators `->` and `*` are defined on smart pointers to transparently reference objects that might only be in the external database prior to the reference. The operator

`=` and constructors on the pointer classes are used to manage reference count and LRU bookkeeping information. The operators `new` and `delete` are specialized on the persistent object classes to implement resource allocation of objects.

To implement persistent objects, there must be a uniform way of referring to objects in the database. This is done by introducing a new data type for objects: ID's. An ID can be translated into a pointer after the referent object is brought into memory. Currently an ID is a 32-bit integer. An ID is further subdivided into some bits of class information, from which the identity of the database file can be determined, and some bits of object ID within that file. Objects stored in the database can refer to each other by means of such object ID's. Other ad-hoc cross-referencing mechanisms can also be used in an application-specific manner, but the object ID provides a unique handle on each object in the database, as well as a convenient key that other objects and clients can use to refer to that object.

All pointers to persistent objects, whether in memory or in the database, are actually pointers to *surrogates.* A surrogate is an object that is used to reference another object.

When an object is first accessed, it is read in from the database. The object's surrogate is modified to point to the actual object, and the surrogate is flagged to indicate that its object is in memory. There is a hash table that maps from object ID's to the address of the surrogate; references can be resolved by using this hash table. When an object is brought in from the database, its references to other persistent objects are replaced by references to their surrogates—if such an object has already been referenced, the surrogate exists, and if not, the surrogate is created.

The operators `->` and `*` are coded as inline member functions on the smart pointer classes to use surrogate objects and to call methods for swapping in the nonresident objects

Although surrogates require an extra indirection compared to ordinary object references, this overhead is acceptable because access methods can be coded inline for the fast case (object already in memory), and because the persistent-object-class writer can determine the granularity of objects that will be stored in the database. In particular, a persistent object may turn into a complex data structure in memory which has only one persistent handle. In this case the programmer must take care with non-smart pointers to the internal components of the structure.

One difficult area in designing persistent object systems is to describe the manner in which data structures are to be *flattened* for database storage and retrieval. We adopted a method-driven approach. When a new class of persistent object is defined, methods for storing and retrieving it must be defined. Though more complex to use, this approach provides a number of benefits. The application designer can tune the representation of objects in the database, possibly distributing them over multiple files or relations, and possibly condensing a large in-memory data structure into one or a few database records. The designer can also put more keys into the database representation of objects, allowing associative access.

This layered approach has proven effective. The initial POS system was implemented on top of a simple sequential access database (NetISAM and C-ISAM), but we have also ported the Cadillac environment to an object-oriented database whose implementation is based on paging and relocation rather than persistent objects and smart pointers.[6] This port required only a few days effort.

7. Protocols

Cadillac runs as a set of Unix processes in which the tool processes are connected through Unix sockets to the kernel.

Protocols are used by tools and the kernel to communicate. A message-passing protocol is the implementation layer for a series of tool-specific protocols that define particular communication activities and, to some extent, the formal definition of tools: any tool that follows a particular protocol is taken to be an implementation of such a tool.

Protocols are defined in layers—higher layers both inherit protocols from more fundamental layers and are implemented on top of them. The lowest layer is the *tool communication protocol*, which specifies how to transmit data from the kernel to the tools. Each class of tool (editor, compiler, etc.) defines its own set of requests that can be transmitted by using the protocol. The set of requests describes what operations the tool can perform as a service for the kernel, and what queries the tool will forward to the kernel. Each request also describes what objects will be transmitted to the tool.

For example, some editor services are *display some text*, *propose operations on annotations*, and some queries are *list the possible operations on an annotation*, *run an operation on an annotation*; and some compiler services are *compile some text*, *check some text for syntax errors*, and some queries are *transmit an include file*, *create a new compiled object*, *annotate some source text*.

The communication between the kernel and the clients goes through a tool interface—a set of routines linked into the kernel—that generates the requests for services and answers queries from the tools. At the other end of the communication stream, each tool must have a *protocol interpreter*, which is a set of routines that answers requests for services and emits queries to the kernel.

[6] At least one commercial OODB firm uses a scheme similar to persistent-objects/smart-pointers.

The number of different tools that can be hooked up to the kernel is limited by the operating system. Implementing a new tool of a known class—a class for which a tool interface exists—is simple: only a protocol interpreter needs to be written for the tool. On the other hand, implementing a brand new tool requires designing and implementing a new tool interface that must be linked to the kernel. This process will be simplified in later Cadillac implementations.

The protocols are patterned after the X11 protocols, so that most messages are asynchronous, some messages can be defined to be received in an arbitrary order (this allows freedom for tool protocol interpreter writers), and the protocols are suitable for use over networks. The following sections describe a sampling of the protocols. Three high-level protocols are described first (compiler, editor, and debugger) followed by two lower level ones (setup and interaction).

7.1 *Compiler Protocols*

The compiler protocol allows Cadillac to acquire dependency information about Cadillac-maintained software components without the kernel having direct knowledge of the language. The compiler acquires source code from Cadillac by using the source code protocol and in return, supplies Cadillac with a high-level view of the language constructs (functions, variable, classes, etc) by using the *language element protocol*. Although the protocol has been designed to handle C and C++, most procedural languages can be projected onto the protocol with only minor extensions. A *command protocol* enables Cadillac to control the execution of the compilation, including incremental compilation.

Because the kernel maintains dependency graphs (one for implementation dependency and one for interface dependency), when part of a large program is altered, only those functions are recompiled that must be. This is called *incremental compilation*. There are two means of achieving incremental compilation. The first, and simpler, method is for the compiler to support *reading* mode, which records declarations but does not analyze bodies or generate code. The kernel requests the compiler to enter this mode, all toplevel forms up to the first toplevel form that requires compilation is processed in reading mode, the kernel requests the compiler to enter *compiling* mode, and the toplevel form to be compiled is sent. Reading and compiling modes are toggled this way for all remaining toplevel forms that require recompilation.

The second method requires the compiler to maintain a data structure that can be thought of as a series of symbol tables, one for each toplevel form. This data structure must be indexed by toplevel form. Also, there must be simple identifiers (ID numbers, for example) to name this data structure and its indexes. When a toplevel form is recompiled, the kernel sends the data structure identifier and index so that the compiler can restore its internal state to that which existed just before compilation of the toplevel form to be recompiled.

In both cases, the compiler produces a `.oi` file, which is an incremental `.o` file. This file is then dynamically loaded into the running image and the original `.o` file for the program or library is updated to reflect the current sources. To be a little more long-winded, all source files and associated `.o` files are maintained in the native (Unix) file system. The `.o` files are kept exactly in sync with the source files, so when an incremental compilation is performed after the source is altered, the incremental loader also modifies the `.o` files.

Cadillac supports compilation which consists of more than one process. Examples include C compilers in which the pre-processor is a separate program whose output is piped to the backend part of the compiler, and language processors such as Ratfor or Cfront which compile into another source language which then needs to be compiled. In such cases, both processes communicate with Cadillac using the compiler protocol (or some subset of it) and Cadillac is responsible for merging the information from the two processes.

7.2 *Text Editor Protocols*

The text editor is the primary user interface to the annotation system. The text editor supports building and activating annotations among objects.

The text editor protocol separates the semantic part of annotations, which is handled by the kernel, from the user interface part, which is handled by the text editor. That is, by using the protocol, the text editor can request the kernel to perform an operation, providing only an operation name. The semantics of these operations are not built into the text editor. The protocol defines how to describe the textual extents of annotations along with a mechanism for how to obtain the names of operations associated with the annotations.

We assume that the following services are provided by the text editor:

- Presenting text in multiple buffers to the user.
- Allowing the user to modify the presented text and save it when desired.
- Displaying special markers in the text buffers that indicate active regions in the buffers.
- Displaying menus associated with different buffers and different active regions, and informing the kernel when items are selected in these menus.

- Informing the kernel when text in buffers or active regions is modified or unmodified because of undos.
- Allowing the user to cut, copy, and paste text together with its active regions.
- Handle simple dialogues with the user.

To implement the text editor protocol, the text editor must be able to do the following:

- Insert graphic markers (or special, non-editable characters) in the text.
- Display extents using different graphic attributes (e.g. different color or different background).
- Display and accept choices (e.g., by using menus) on request.

To accommodate various text editors, the protocol loosely defines the graphic objects and attributes that should be used for extents.

The text editor receives text as an array of characters together with extents. Extents correspond to annotations. The editor must be able to support a dialogue with the user to determine which annotation methods to invoke.

The editor also supports a protocol to determine which zones of text have been modified to support incremental compilation. This is a good example of the style of interaction and operation in Cadillac.

Suppose the user types some new code into an existing C++ function. The editor issues a *zone-modified* message to the kernel along with the zone identification. When the kernel receives the message, the dirty C++ function is immediately reprojected in a different font. The dependency registration protocol is used to register dependence between tools and objects, so that all recompiled functions that have dependent tools will be notified to reproject themselves after recompilation.

7.3 *Debugger Protocols*

The cadillac debugger protocols are designed to allow the kernel to interact with sequential debuggers. We assume that debuggers are able to do the following:

- Control the execution of a program: start, stop, resume, etc.
- Report the source file position of a stopped program.
- Get information on local and global variables: type, value, address, and position in the source file.
- Modify the value of variables.
- Accept commands typed by the user and provide textual responses.

The debugger protocols are easily expressed in terms of the command set for the typical Unix debugger, so the initial version of the debugger is simply an encapsulation layer around the GNU debugger (similar in spirit but not yet as general or extensible as the HP Encapsulator). This encapsulation layer interacts by receiving messages from the kernel, constructing debugger command lines, parsing the output, and sending messages back to the kernel.

This approach has allowed us to get acceptable integration of the GNU debugger with Cadillac without making any changes to the debugger code. This encapsulation facility will be generalized.

7.4 *Setup Protocols*

This protocol is used by a tool or by the kernel to establish a connection between them. A socket is created and a dialogue ensues during which the tool identifies the protocol it wishes to follow and the kernel revision level it was implemented for. Once the dialogue is completed successfully, the kernel can begin to interact with the tool.

Certain tools can be used from the shell, also. It is common for C++ programmers to use `make` files. If `make` starts up the compiler with the `-cadillac` flag, the compiler connect to the kernel which filters compilation requests based on its dependency graphs. This way, `make` will compile only those portions of the source that require recompilation, and the `.o` files will also be incrementally updated.

7.5 *Interaction Protocols*

This protocol is used to encapsulate the common parts of all interactive protocols. Such protocols support display of objects and their annotations, and they support user interaction to query annotations and invoke them. Typically objects register their dependence on all available interactive tools, so that they are automatically reprojected when they change.

This dependency registration protocol along with the use of the interactive protocol implements the behavior of each tool appearing to know about the others. For example, if one window contains a text editor buffer with some source code and another a grapher with a different projection of the same source, the user can recompile from either window—because the same objects are projected into each with along with the same annotations (and hence with the same meanings). The grapher can be used to navigate through the source graph and the text editor window can follow along. Editing the source code to use other classes causes the grapher window to change after recompilation.

7.6 *Inheritance of Protocols*

The protocols are organized into a hierarchy to use inheritance. For example, the debugger inherits the setup protocol, the interaction protocol inherits from the setup protocol, the grapher and editor protocols inherit from the interaction protocol (and hence from the setup protocol), and the compiler and macro processor protocols inherit from the setup protocol.

8. User Interface

The nature of Cadillac is for tools to annotate source code and for presentation clients to present views of the source code with annotations. For example, compiler diagnostics are represented as annotations on the source text. When the compiler runs and finds an error, the source text is annotated with the error. This annotation is highlighted in some way (currently with a glyph to its right). When the user requests Cadillac to follow the annotation, a temporary window is created containing the series of diagnostics with a visual link between the diagnostic and the corresponding source code. This temporary window can be made permanent. Also, each diagnostic in the series is annotated with its source.

Similarly, performance data can be shown as annotations on the source text; dependencies determined by the compiler are represented as annotations. As much as possible, user interaction is through annotations on source text, which may be projected as a graph. The visible annotations can be filtered to show only information of a certain type. Annotations are instances of classes, and each class can be thought of as a relation, and therefore the user can select which relations to view.

This style of interface is unlike the "thousand windows" style, because Cadillac tries to put as much information as possible into each window; this information is not always fully displayed, but can be abbreviated or elided, and the connections between this information and what it relates to is also visually apparent. This allows the programmer to not be bothered by a plethora of windows unless it is necessary or desired.

9. Comparison to Field and SoftBench

The most closely related environment architectures are Field and SoftBench.

Each of these other environments use a message-passing model. In Field each tool registers patterns of messages, and the kernel resends to each tool any message that matches its patterns. In SoftBench the tools have protocols that they follow according to their tool types, and the kernel sends messages of a certain type only to tools that request messages of that type.

In SoftBench all the tools are required to send notification messages at the end of each action. This lets other tools set things called *triggers* to go off. Field supports a similar protocol. SoftBench supports an extension language in which triggers can be written.

The kernels of Field and SoftBench act primarily as message routers. Each tool is required to register its interest in the activities of other tools, and so the control is distributed among the tools.

Cadillac provides similar facilities through its use of tool protocols and the dependency protocol. Cadillac stores all objects in a POS so that the state of the development process is captured. All control and data integration and user interaction is through a single mechanism—annotations—that simplifies effective utilization. Knowledge about the various facets of programming are isolated in the tools. For example, all knowledge about C++ syntax resides in the compiler and is accessible through the compiler protocol.

Cadillac achieves its tight integration by providing the equivalent of a monolithic environment. However, very large portions of that environment are implemented using abstractions whose implementations are remote and available only through the protocols. That is, the environment *has* a compiler that it uses, but the implementation of that compiler is not part of Cadillac; only the abstract interface to it is in Cadillac.

10. Future Plans

Cadillac currently provides a general hypertext system that can be used to link code and documentation. Tools and their interfaces are needed to support an integrated coding/documentation system. Automated testing can be supported by using annotations. Better encapsulation facilities are required to allow quick protocol adaptation of tools. An extension language is needed to be able to add new protocol servers to the kernel. The user should be able to define new annotation and object types, both through an extension language and graphical dialogue.

Appendix A. Glossary

To describe the Cadillac architecture we make use of the following terms:

Server: A program that provides service or functionality to a variety of other programs called clients through a message-passing medium such as byte streams in Unix. A server rarely performs user interface activities.

Client: A program that communicates with a server for the purpose of information exchange. A client might engage the user in a dialogue. Though the terms client and server are arbitrary, a client generally communicates with relatively few other programs while a server generally communicates with many other programs.

Kernel: A server that performs the fundamental actions in an environment.

Protocol: A language designed to pass data and control between a client and a server. Typically a protocol is a series of messages and containers of information encoded in a common medium, such as byte streams or ASCII text.

Instance of class A: An object o whose class is A and $\forall C$, $C \subseteq A$, $o \notin C$. Some call this *direct instance* ($\subseteq$ denotes the subclass relation).

Member of a class A: An object whose class is A or a subclass of A.

References

[**Cagan 1989**] Martin Cagan, *HP SoftBench: An Architecture for a New Generation of Software Tools*, SoftBench Technical Note Series, SESD-89-24 Revision: 1.4, November 1989.

[**Gerety 1989**] Colin Gerety, *HP SoftBench: A New Generation of Software Development Tools*, SoftBench Technical Note Series, SESD-89-25 Revision: 1.4, November 1989.

[**Reiss 1989**] Steven P. Reiss, *Interacting with the Field Environment*, Brown University Technical Report No. CS-89-51, May 1989.

CADILLAC

SOFTWARE DEVELOPMENT ENVIRONMENT

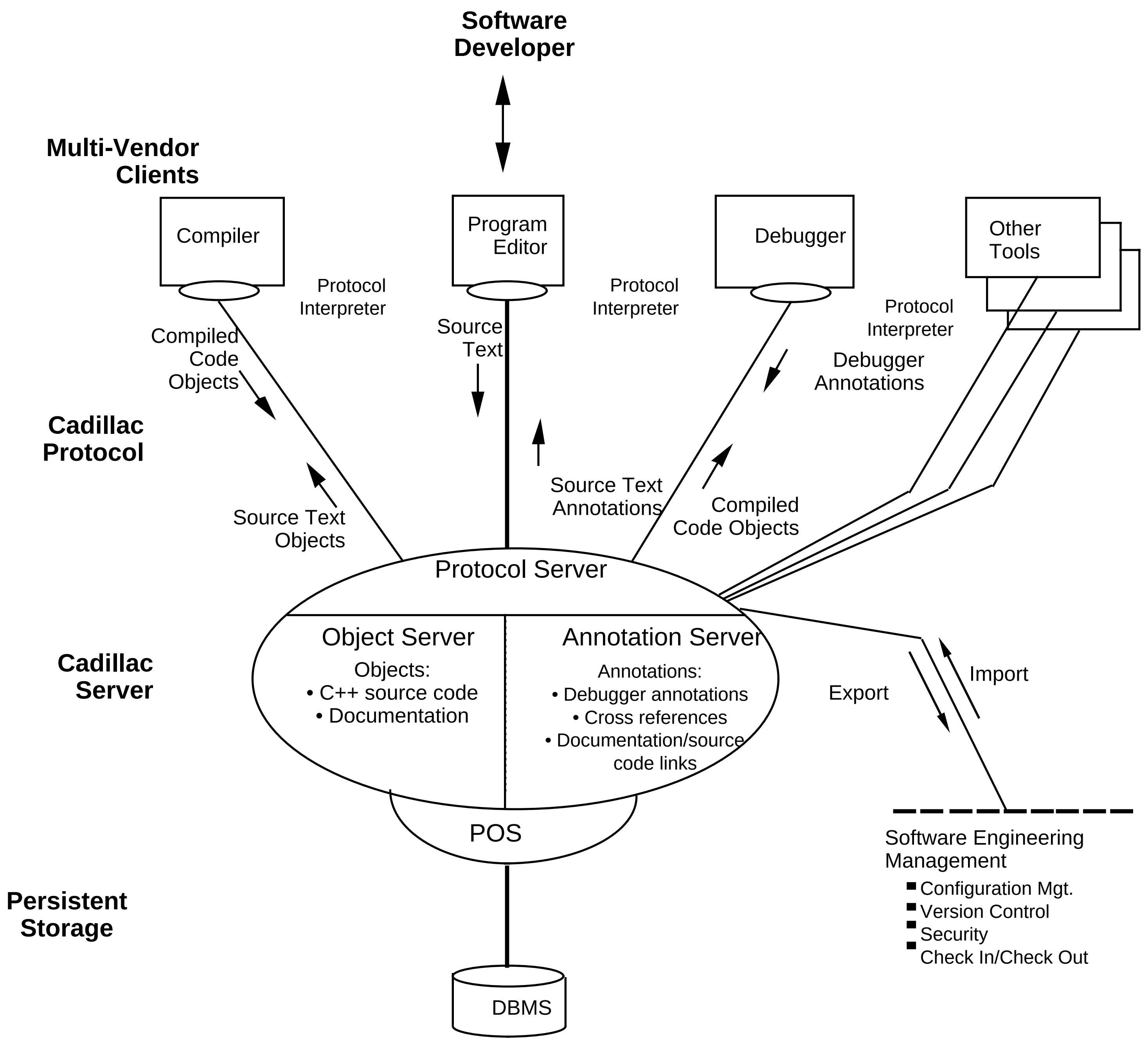

Figure 1