further languages being added as necessary.

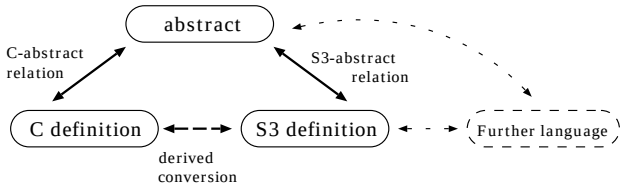

Figure 6: Mixed-language programming

Notice that due to the use of encapsulated methods, our requirement that existing tools still be supported is met. By "hiding" the workings of the conversions beneath the abstract definition, a user of the database not concerned with mixed language working can continue to use modes as before, as the use of a mode in an S3 situation will extract the relevant field.

5 CONCLUSIONS

The corner-stone of the project is Active Objects. The increased flexibility this will provide within the existing software development environment will not only facilitate the enhanced CADES methodology and mixed language work, but will allow tools which need to access databases to be written more easily. Further, our decision to use an object-oriented language as the basis for Active Objects means that such tools can be designed and implemented more reliably and with maximum reuse of software.

Eiffel has turned out to be an excellent choice for the Active Objects Processor implementation language. Multiple inheritance has allowed us to separate the definitions corresponding to fields stored on the database from active fields, providing the possibility to reuse the latter.

The Eiffel language provides an interface to C which can be used to invoke existing tools. The Eiffel environment can generate C source code from an Eiffel library which can be invoked from existing tools. These interfaces are essential for this work, as the Active Objects Processor must be integrated into existing tools.

6 FUTURE WORK

The next phase of the project will be to enhance the S3 systems programming language with object-oriented features. This will increase the power of S3 and will enable software to be produced with less effort and reduced cost.

References

[Horowitz, 1991] M. L. Horowitz. An Introduction to Object-Oriented Databases and Database Systems. Technical Report CMU-ITC-91-103, Information Technology Center, Carnegie-Mellon University, August 1991.

[Hughes, 1991] J.G. Hughes, editor. *Object-Oriented Databases.* Prentice-Hall International, 1991.

[Kernighan, 1988] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall International, 2nd edition, 1988.

[Meyer, 1988] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall International, 1988.

[Meyer, 1992] B. Meyer. *Eiffel: The Language.* Prentice-Hall International, 1992.

[Pearson, 1973] D.J. Pearson. CADES – computer aided development and evaluation system. *Computer Weekly*, 1973.

[Purtilo, 1991] J.M. Purtilo and J.M. Atlee. Module reuse by interface adaption. *Software — Practice and Experience*, 21(6), June 1991.

4 SUPPORT FOR MIXED LANGUAGE WORKING

One of the areas covered by this project is the support of mixed language working. The idea here is to allow a programmer to mix pieces of code written in different programming languages without being concerned about the language in which a called routine is written. Thus in a piece of code, the function or procedure call `foo(x)` can be used regardless of the source language in which `foo` is implemented. Attempts have been made in this direction (such as [Purtilo, 1991]), but not in a way which copes transparently and flexibly with calls between a number of different languages.

The benefit of allowing this is obvious. Reuse of existing code becomes easier, as we are no longer restricted to using the original language. In the application under consideration, the CADES database contains a large amount of S3 code, which we wish to be callable by those using languages other than S3, in particular C. However, the approach used is applicable to combinations of other languages, or even two different implementations of the same language.

The difficulty with mixing languages is the passing of arguments and results between routines. Take the example of `STRING`s, for instance, which are intuitively sequences of characters. Consider the following piece of C code:

```
c_rout(STRING s, int x)
{
    ...
    foo(s);
    ...
}
```

where `foo` is a routine written in S3, expecting a `STRING`. The problem is that strings have a different representation in S3 than in C (in C, a string is implemented as a pointer to a null terminated area of memory, whereas in S3, a *descriptor*, containing type and size information as well as a pointer is used), so we must perform some conversion on the argument before passing it to the routine. After running the code through the Mixed Language Processor, something like the following would be expected:

```
c_rout(STRING s, int x)
{
    ...
    Desc desc;
    desc = convert_c_string_to_s3(x);
    foo(desc);
    ...
}
```

Having noted that `foo` expects a parameter of type `STRING` (this will be recorded in the holon interface owned by `foo`), and that `foo` is an S3 holon (again information obtained from the database), the Mixed Language Processor has inserted the necessary conversion procedure to ensure that the argument passed to `foo` is now in S3's `STRING` format.

The approach taken to solve this problem is to alter the way in which modes or types are stored on the database, adding an extra layer of abstraction. At present, a mode record has two main fields, one recording the name of the mode, the other with the S3 definition.

We add some extra fields which provide a description of the type, independent of the actual representations used in the programming languages. The language used to describe this "abstract" description will contain constructs such as seq of and basic types like character or integer. We also need to store the actual representations in the programming languages, as before, but in addition, we also store the relationships between these programming language representations and the abstract description.

From these relationships, we can derive an algorithm, or method, for converting an instance of a type in one representation to an instance in another representation. So in the example above, we would produce an algorithm which takes an S3 descriptor, and returns a C pointer to an appropriate area of memory. Although this method is derived via the relationships stored, it is considered to be a field on the record, and is thus an example of an "Active Field", as a request for retrieval of this field can result in the invocation of some computation.

A further benefit of using this approach can be seen if we wish to add further languages. If conversions were "hard-wired" into the system, for each language we would have to add conversions to and from all other languages currently supported. If conversions are derived via the abstract type, then only this relationship need be added, with the conversions between languages derived.

The situation is illustrated in Figure 6, with

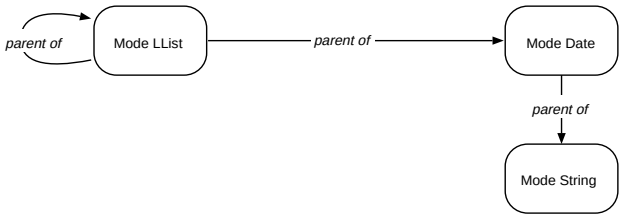

Figure 4: *Modes* in CADES

Next, a function can be defined which operates on a Date (in fact it prints one out in the form *January 13, 1992*). Note that `printf`, a function which prints text, is part of the standard C library.

```
void print(Date d)
{
  printf("%s %i, %i\n",
         d.month,
         d.day,
         d.year);
}
```

This will result in the addition of two new objects to the database, a holon and a holon interface, and links between them as illustrated in Figure 5. It is assumed that a holon interface `printf` already exists, and a usage link is added from the holon `print` to the `printf` interface.

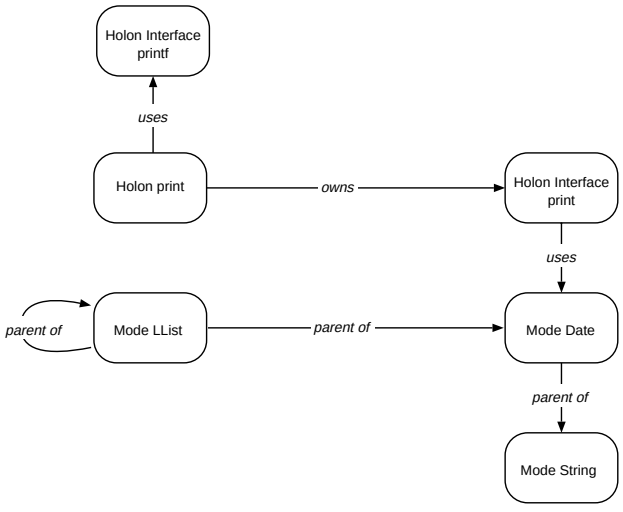

Figure 5: Holon and interface objects

If a further function to print out a list of dates – `printList` – which makes use of `print` were defined and entered into CADES, then two more objects (a holon and a holon interface) would be added, along with the same sort of usage (from the holon and interface to the modes) and ownership (from holon to interface) links as above. In addition, a usage link would be added between the `printList` holon and the `print` holon interface, to record the fact that `printList` makes a call to `print`.

A program called an *Environment Processor* exists which, given the name of a holon, generates a file containing the text of the named holon, along with all the subsidiary definitions required by that holon (for example, all the type definitions which it uses). This file can then be passed directly to the C compiler.

Controlling and managing the design work and code which is produced in large software development projects has long been recognised as a problem, and many attempts have been made to address it. Developers in the Unix environment often use tools like `sccs` and `rcs` for maintaining a number of different versions of a product – CADES addresses this problem too, and can store multiple versions of objects and links. The Unix `make` utility fulfills a similar role to CADES in that it records dependency information, and the method by which an entire product or system is constructed from its component parts. The CADES approach has a few advantages over Unix's `make` and similar tools. Firstly, CADES enforces its philosophy more than `make` does, so that users are effectively forced to split code up into small units, and record all the dependency information as links (numerous tools exist to help with this kind of thing). This means that source code and dependency and construction information is stored in a very uniform way, which is not always the case on Unix systems. Secondly, the dependency and linking mechanisms in CADES record more information than `make` does – not only is the fact that some entity a depends on some entity b recorded, but also information about the nature of (or reason for) this dependency (e.g., does a use b, or is a a parent of b). This means that speculative queries can be made of the CADES system (for instance, "what happens if I change this interface"). In an environment where many people are working on different aspects of the same development, or different subsystems of the same product, such dependency tracing facilities must be available if one is to have any faith in the reliability of the design and maintenance process.

3 CADES METHODOLOGY

Here we discuss the way in which the CADES database [Pearson, 1973] is used as a repository for source code objects, and to capture design information. An example is given showing how CADES might be used in an environment where source code is written in the C language [Kernighan, 1988]. CADES was originally (and still is) used to store code written almost exclusively in the S3 language, but the data model is sufficiently flexible to support many imperative programming languages with relatively little change. Some of the current effort is directed towards providing support for C programming in the CADES environment, and future work will look at supporting object-oriented programming languages.

Entities in CADES are typed and named, and have fields (whose values are strings or numbers). The types of objects which exist in a particular instance of a CADES-like database are defined by a *schema*, as are the fields which each type of object can have. There are four basic types of object in the schema of the CADES system, and these correspond to the main elements of imperative programs. *Mode* objects are used to represent data type definitions, *Data* objects represent global variable declarations, *Holons* represent the bodies of procedures or functions (i.e. blocks of code), and *Holon Interfaces* represent headers or signatures of procedures or functions. Mode, Data, and Holon Interface objects all have a field called *Decl* which stores the text of the item's declaration, and Holon objects have a field called *Body* which stores the text of the procedure's code[4]. The separation between procedures and their interfaces may, on first sight seem unwieldy and overly complex. The justification for the split is founded in the belief that a procedure's specification should be represented separately from its implementation. The interface represents the 'external' or 'customer' view of the procedure, whereas the holon object stores the actual implementation (which should not be of interest to customers).

As well as storing collections of objects, CADES maintains links between them which identify usage and dependency relationships, as well as grouping information which serves to document aspects of the development process. The two most important classes of link are known as '*uses*' which record information about potential executions, and '*parent of*', which records more static structure. For example a holon is linked to a holon interface by a 'uses' link if the holon's code contains a procedure call to the interface, and if the definition of one data type mentions the name of another, the first will be a 'parent of' the second. An '*owns*' link is used to connect a holon and its corresponding interface.

The main motivation for storing units of source code as nodes on a database, and documenting dependencies as links, was to aid configuration management. Tools exist which traverse the database, following usage and other links, collecting together the code associated with nodes to construct entire programs which can be passed to the relevant compiler. The links allow the effects of changes and modifications to be tracked, so that when an object is changed, everything which depends on it can be checked for consistency, and possibly edited, reconstructed, or recompiled. Before changes are made to the database, the impact of these changes can be investigated. A further advantage of managing source code in such a 'fine grained' way is that it encourages code reuse by storing code in small units and maintaining all the dependency information.

The following example illustrates how the database nodes and links are used by considering some simple definitions in C. First, a type of linked list structures is defined, where the elements of the list are dates.

```
typedef struct LList {
        Date item;
        struct LList *next} LList;
```

where dates are defined as

```
typedef struct {
        int day;
        String month;
        int year;} Date;
```

These definitions are added to the database as two Mode objects, with a 'parent of' link between them, and a 'parent of' link between `Date` and the mode `String` which is assumed to exist already (see Figure 4). Note that, because of the recursion in the definition, a circularity exists in the parent of links (`LList` is a parent of itself).

[4]Actually, Holons don't really have a Body property at all. The text is stored in a file and Body is an *active field* which returns the contents of the file.

```
    DOD: DATE
    update_DOD(s: DATE)
    NO_OF_BOOKS: INTEGER
    AGE: INTEGER
    update_AGE(i: INTEGER)
    WROTE: LINKS[BOOK]
    new(nm: STRING)
end
```

Each field of the database record has up to two Eiffel features associated with it — one for retrieval and one for update. Though we require this for implementation reasons (we need to register which fields are updated so that we can efficiently transfer changes made by the application to the physical database) it is also required by the Eiffel language, in order to hide from the client which features are actually implemented as attributes.

Some fields, like *NO_OF_BOOKS*, may not be updatable. The implementation of the non-active fields simply accesses the database on which that field is stored. The implementations of active fields (such as *AGE*) are Eiffel procedures and functions. For example *AGE* might be implemented as follows:

```
AGE: INTEGER is
do
    if DOD=void then
        Result :=
            current_date.year - DOB.year
    else
        Result := DOD.year - DOB.year
    end
end
```

Links between records are features of the generic type *LINKS[X]* where *X* is the record type at the end of the link. This class is essentially a linked list of *X*. Features of this class allow links to be added, deleted and accessed as well as providing an iterator to loop through the linked records. Since these operations have side-effects on the link, we provide a *view* of the relationship via the class *LINKS[X]*. This is particularly important since active fields themselves may make use of links, and this fact should not be visible to the client.

An example implementation of the field *NO_OF_BOOKS*, which returns the number of *WROTE* links from an *AUTHOR* will clarify this.

```
WROTE: LINKS[BOOK] is
-- implementation not important

NO_OF_BOOKS: INTEGER is
local
    w: LINKS[BOOK];
do
    w := WROTE;
    from
        w.start
    until
        w.offright
    do
        Result := Result+1;
        w.forth
    end
end
```

We take a *copy* of the link object in order to traverse it. In this way we avoid any unwanted side effects becoming visible to the user of *NO_OF_BOOKS*, who may themselves be in the middle of examining the same links.

2.5 Other Issues

Other issues that are not central to the above discussion, but may be of interest to some readers are discussed briefly below.

Insulation between tools written in Eiffel and the Active Objects Processor. If an interface to the Active Objects Processor classes should change (as a result of adding a new field for example) the tools written in Eiffel would have to be re-compiled if they used the Eiffel classes directly. This is unacceptable as there may be many tools. One solution to this problem requires tools to use "look-alike" classes which communicate with the Active Objects Processor over a static interface, enabling the run time linking and storage management systems to be isolated.

Remote access to the Active Objects Processor. A requirement placed upon the design for Active Objects was that it should allow for the possibility of access from client workstations. This will be achieved by providing a variant of the insulation mentioned above which invokes remote feature access rather than local. In simple terms, the static interface includes a communications link.

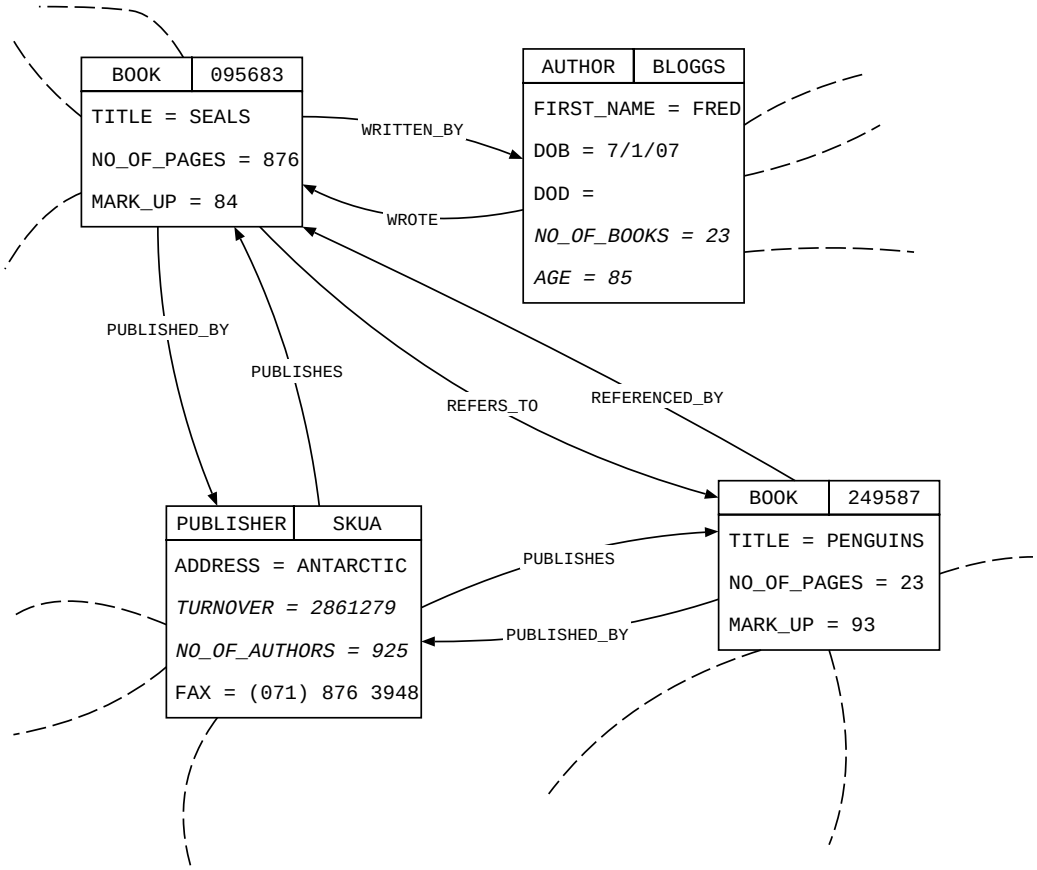

Figure 2:

Active Objects Processor decides where the requested information is stored and then accesses that database. Note that from the user's point of view there appears to be only one database.

2.4 The Active Objects Processor

The implementation language for the Active Objects Processor is the object-oriented language Eiffel [Meyer, 1992]. We chose an object-oriented language as the record structure of the database fitted in well with classes. Database fields are then methods on these classes which access the appropriate database. Also active fields are methods on these classes which access and manipulate the values of non-active fields. In other words an object-oriented schema corresponding to the CADES database schema can be produced [Hughes, 1991]. Issues involved in the design of object-oriented databases are discussed in [Horowitz, 1991].

The use of a general-purpose, computationally complete language has the advantage of giving an interface to the databases which is more easily used within a program than DNL. User interfaces can be written in that language using the classes which represent the databases directly, as illustrated in figure 3. Also, a full object-oriented language means that code can be reused in several tools if desired.

Eiffel was chosen because it provides the programmer with a more formal framework than other object-oriented languages [Meyer, 1988]. Strong typing and the use of preconditions, post-conditions and class invariants mean that tools interfacing to the Active Objects Processor are more reliable.

Below is the Eiffel class interface to the `AUTHOR` database record.

class interface *AUTHOR*
creation
    new
feature specification
    FIRST_NAME: STRING
    update_FIRST_NAME(s: STRING)
    DOB: DATE
    update_DOB(s: DATE)

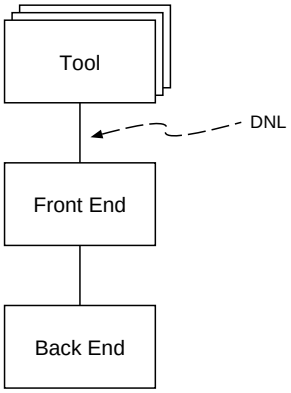

Figure 1:

These interfaces use a textual database query language called DNL. [3] This language, which will not be described in detail here, allows users and tools to "navigate" around a network of linked records and access fields on these records. Records are uniquely identified by type and name. Fields and links between records are identified by name.

Figure 2 illustrates a fragment of a bibliography database. There are three record types in this database — `BOOK`, `AUTHOR` and `PUBLISHER`. `BOOK`s are identified by their ISBN number, `AUTHOR`s by their surname and `PUBLISHER`s by their name. Fields valid on `BOOK` records are `TITLE` and `NO_OF_PAGES`. `BOOK` records are linked to `AUTHOR` records via `WRITTEN_BY` links, `PUBLISHER` records via `PUBLISHED_BY` links and so on. Note that for each link there is a corresponding link in the reverse direction. Fields in *italics* are described below.

2.2 Active Fields

Given this structure and software architecture the problem is to enhance the database with "active fields". An example of an active field is the *AGE* field of an `AUTHOR` record. The value of this field is not actually stored on the database but is calculated from the `DOB` and `DOD` fields. If the `DOD` field has no value then the age of the author is calculated using the current date. Updating the *AGE* field will result in the `DOD` field being updated.

This idea allows some of the tools which interface to the front-end to become active fields and so be accessible from DNL. Also, fields which are currently inactive, like `DOB`, can become active without the interface to the front-end changing. This is essential if representations are to change leaving the interface used by tools and users unchanged. For example, we may decide to store a date in microseconds and calculate the more familiar string of characters (`7/1/07`) from it. This kind of feature is required for the mixed language work, see section 4.

[3] Database Navigation Language

2.3 Proposed Software Architecture

Figure 3 describes the proposed architecture which facilitates active fields and maintains the DNL interface so existing tools still work.

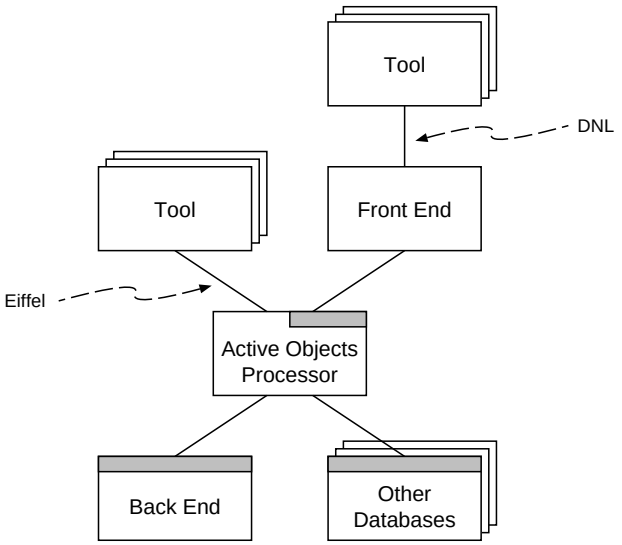

Figure 3:

The "Active Objects Processor" takes requests which would normally be passed to the Back End, evaluates the active field requests and calls the Back End with the non-active field requests. To incorporate an existing tool in the Active Objects processor, we first define the output of the tool as a new feature within the user view of the database. The logic of the tool is then included in the active objects processor as the implementation of that feature. The user interface aspects of the tool however remain separate, which is why there is still a 'Tools' box in the new architecture. This is in fact a classic client-server architecture, and will enable us to exploit new workstation technology by moving the user interface part away from the mainframe.

By extending the database schema used by the front-end, requests to access fields and records not held on CADES can be generated. The

means of access to the repository, being used as both a user interface and a tools interface.

The tools currently only support two languages directly, S3 and SCL. S3 is an Algol68 derivative which was developed at the same time as CADES for the purpose of writing VME. SCL is the system control language for VME, and is not unlike S3. Support for other languages such as C and COBOL is present but without the finer grain representation, the benefits of which are discussed in section 3.

1.2 Assessment

To date this architecture has served us well. It has enabled us to maintain and develop VME over a period of 20 years with few of the problems of structural decay normally associated with software engineering on this scale.

During the same time, however, the fundamental CADES tools have changed little. Extra tools have been produced for specific purposes, and some work has been done to improve general usability, but there has been as yet no attempt to exploit the new software engineering technologies that have been developed since CADES was originally implemented.

Specifically,

- We require full support for newer languages such as C, C++, Eiffel. In particular we need seamless integration of components written in different languages, so that we can take advantage of existing components and components written elsewhere without prejudicing our choice of language.

- In todays open world we can no longer afford to develop products for VME only: we need to be able to produce tailored products for a range of environments from a single source.

- The development environment is still VME based: we need to exploit the potential of distribution to workstations, in particular to provide improved user interfaces and better integration with associated development activities such as document production. By implication the benefits of CADES will become available to non-VME developers.

1.3 The Project

The objective of the project[2] is to improve software development productivity. We intend to do this by introducing object oriented technology, both in the tools themselves and in the supplied development processes. This will be done in an evolutionary fashion: one of the prime requirements is that the existing tools and methods must continue to be supported.

We describe the first step as 'Active Objects', described in detail in section 2. The current CADES database contains only base data: any derived results are produced by running tools, which present their results via some arbitrary interface. We will provide the means whereby these results can themselves be represented in the CADES schema, such that their retrieval will cause the appropriate tool to be executed. Besides providing a clean and consistent user interface, this also enables us to replace existing data by some other form without impacting existing tools. A particular application of this is seen in the mixed language support, section 4, where we replace the existing language specific declaration of interfaces by a more abstract form from which the original declaration can be reconstructed.

Other benefits of this work are that we will provide a programming language interface to the database as an alternative to the textual one, and that we can include databases other than CADES in the schema we present to the user. The choice of the vehicle for this (Eiffel) is discussed in 2.4.

2 EIFFEL AS A DATABASE INTERFACE LANGUAGE

2.1 The Current Architecture

The current structure of the database software, as described above, is shown in figure 1.

The "Front End" provides interfaces for:

- interactive database queries;
- interactive database updates;
- tool invoked database queries; and
- tool invoked database updates.

[2]Supported by the Science and Engineering Research Council and the Department of Trade and Industry through the Teaching Company Scheme.

Improving a software development environment using object-oriented technology

M.A. Firth
M.H. O'Docherty
R.E. Fields
S.K. Bechhofer
T.C. Nash

ICL Corporate Systems Division
Wenlock Way
West Gorton
Manchester M12 5DR

University of Manchester
Department of Computer Science
Oxford Road
Manchester M13 9PL

13 March 1992

Abstract

This paper summarises the work of a University of Mancshester / ICL collaborative project, which is charged with "introducing evolutionary enhancements to ICL's development environment to benefit productivity and time to market". This applies in particular to the tools used to develop and support VME, ICL's mainframe operating system, though it is our intention to make the work as general as possible.

We describe the use of the Eiffel object-oriented programming language as an interface to an existing database system, a design repository. We give a brief overview of the history of the system to explain the reasons for adopting the new technology, and discuss the expected benefits. We present the reasons why we chose Eiffel as the vehicle rather than any other language or a proprietary solution.

We discuss the benefits of holding implementation level information as nodes in a network database instead of as discrete files, and show how the system will be evolved to support multiple implementation languages by exploiting the new object oriented interface.

1 INTRODUCTION

1.1 History

The current development environment is based on a design repository called CADES[1] which has its origins in the late '60s / early '70s. It was designed specifically to support the development of the VME operating system for what was then the 'New Range' or 2900 series of mainframe computers.

At the lowest level CADES is a network database. On top of this there is a layer known as the back end, which provides configuration management and a logical schema view. A further layer, the front end, provides a textual update and retrieval language. Currently this is the only

[1] Computer Aided Development and Evaluation System