This paper should be referenced as:

Kirby, G.N.C., Morrison, R. & Munro, D.S. “Evolving Persistent Applications on Commercial Platforms”. In Advances in Databases and Information Systems, Manthey, R. & Wolfengagen, V. (ed), Springer-Verlag, Proc. 1st ACM SIGMOD East-European Symposium on Advances in Databases and Information Systems, St Petersburg, Russia, In Series: Electronic Workshops in Computing, van Rijsbergen, C.J. (series ed), ISBN 5-7940-0004-X (1997) pp 170-179.

Evolving Persistent Applications on Commercial Platforms

Graham Kirby, Ron Morrison and David Munro
School of Mathematical and Computational Sciences,
University of St Andrews, St Andrews, Fife, Scotland KY16 9SS
{graham, ron, dave}@dcs.st-and.ac.uk

Abstract

This paper draws on the experience gained in designing and implementing the orthogonally persistent programming languages PS-algol and Napier88. They provide, amongst other facilities and within a strongly typed environment, the underlying mechanisms for programming the evolution of persistent application systems. The essence of the paper is to suggest how such provision may be taken from these research platforms and delivered in representative commercial systems such as Java with a persistent store, and O_2C.

1 Introduction

The persistence abstraction allows the creation and manipulation of data in a manner that is independent of its lifetime thereby integrating the database view of information with the programming language view. This yields a number of advantages in terms of orthogonal design and programmer productivity which are beneficial for application systems [6]. Three approaches to providing persistence are possible:

- design and implement a new language and object store;
- add an object store and some persistence facilities to an existing language; or
- add a programming language and some persistence facilities to an existing database system.

In the early days of persistence research, it was hypothesised "that it should be possible to add persistence to an existing language with minimal change to the language" [3]. While the statement turned out to be true, it was not true in the manner that was first expected. That is, "an existing language" turned out to be existentially quantified rather than the intended universal quantification: it was found difficult to add persistence to several existing languages. Thus the preferred early technique for providing persistence was to design and implement a new language with persistence built in. Since then many persistent languages have been developed in this manner including Amber [7], Fibonacci [1], Flex [11], Galileo [2], Napier88 [20], PS-algol [22], Trellis/Owl [24] and Tycoon [18]. All of these must be considered research systems.

In commercial environments, the second and third approaches are most common, due to the costs involved and the need to interface with existing systems. For example, orthogonal persistence for Java [5] takes the object-oriented language Java and adds persistence facilities, and the object-oriented database O_2 [12] provides languages to interface with the database. All three approaches however have a common interest in ensuring that data, programs and meta-data can evolve as the applications systems, and the uses to which they are put, evolve. In order to support evolution of persistent applications a language or database environment must provide a number of basic facilities either within the language/database itself or within its run-time support system. Here we selectively draw on the PS-algol/Napier88 experience and show how, by providing these basic facilities, the evolution of persistent application systems may be programmed within the system. The essence of the paper is to demonstrate how these basic facilities, taken from the research environment, are provided in two representative commercial systems, namely Java with a persistent store and O_2C, thereby yielding the same attendant benefits.

2 Supporting Evolution in Persistent Application Systems

System evolution may incur change to the data, the programs that access the data and the meta-data. Changes to program and data with the invariant of fixed meta-data are normally handled by updates to methods and data respectively. The difficult problem is to change the meta-data while keeping all the existing programs and data consistent with the semantics of the change [10]. This problem we believe is intractable in general but where it is tractable a mechanism is required for programming the evolution from within the system so that the application can evolve itself.

The mechanism used for evolution, described here, is sometimes called type safe linguistic reflection [25]. In it the executing application generates new program fragments in the form of source code, invokes a dynamically callable compiler, and finally links the results of the compilation into its own execution. The generation of the source code is most effective where the values and types of existing objects are used to tailor the generated code to their specific needs. Thus by having data, programs and meta-data available for manipulation within the application, new programs, data and meta-data can be generated, based on the existing values but altered to accommodate the specified evolutionary

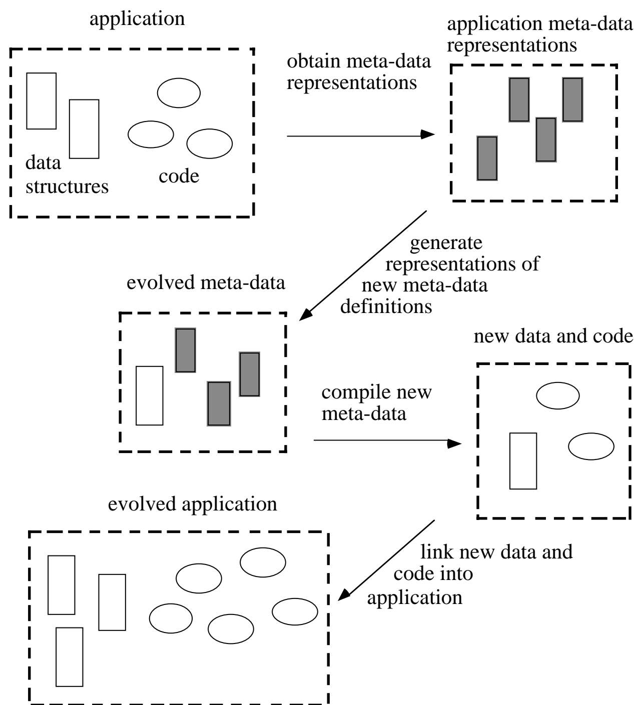

Figure 1: Application evolution process

process. If the evolution steps are performed within a single atomic transaction it may be possible for some applications to continue executing while another is being evolved. Figure 1 illustrates the process.

From our experience with PS-algol and Napier88 the following facilities are considered to be essential to support the evolution of persistent applications:

- a mechanism that provides dynamic descriptions of meta-level information, such as types, classes and source code;
- a compiler accessible by persistent applications;
- an infinite union type with injection and projection operations; and
- a mechanism supporting dynamic incremental loading.

Several other facilities are particularly useful, although not essential in general:

- a structural type equivalence mechanism to allow separately prepared programs and data;
- a persistent store with root(s), reachability and referential integrity; and
- the ability to manipulate code as data, and to store references to objects in the code.

We now turn our attention to how these facilities are provided in Napier88, a persistent language with higher order procedures, and in two representative commercial systems, O_2 with O_2C, and a persistent form of Java.

```
IsType :        proc( TypeName, TypeRep -> bool )
EqualType :     proc( TypeRep, TypeRep -> bool )
Display :       proc( TypeRep -> string )
ScanStructure : proc( TypeRep ->
                      proc( -> string ) )
FieldType :     proc( TypeRep, string -> TypeRep )
Elms :          proc( TypeRep -> TypeRep )
Args :          proc( TypeRep -> List[ TypeRep ] )
Result :        proc( TypeRep -> TypeRep )
```

Figure 2: Napier88 type manipulation procedures

3 Application Evolution in Napier88

Napier88 is a persistent language with structural type equivalence, higher-order procedures, parametric polymorphism and abstract datatypes [20]. The meta-data for a Napier88 application comprises representations of the types of the procedures which make up the application and of any associated data structures, and source code representations of the procedures. The evolution of a persistent application in Napier88 involves the following steps:

- Meta-level descriptions of existing values and types are obtained using the procedures `getType` and `getProcSource` which are supplied within a pre-populated persistent store.
- Based on analysis of the meta-level descriptions, program fragments defining new values (which may themselves be procedures) and types are generated in the form of strings. These are then compiled by invoking the Napier88 compiler to produce first class procedures whose bodies comprise the generated code fragments.
- The dynamically created procedures may then be called in order to execute the generated code.

These steps are now described in greater detail.

3.1 Meta-level Descriptions

Meta-level descriptions are provided in Napier88 by the type `TypeRep` which is used to represent type information [8]. A `TypeRep` instance may be obtained for a given value by calling the following library procedure:

```
getType : proc( any -> TypeRep )
```

This takes an instance of the infinite union type `any` and returns a representation of the type of the underlying value. The value for which the type representation is required must therefore first be injected into the type `any`. `TypeRep` is an abstract data type and so the structure of the representation is inaccessible, however a number of library procedures which operate over type representations are provided, including those shown in Figure 2.

The procedures `IsType` and `EqualType` allow a type representation to be tested for a particular constructor, e.g. structure or procedure, and for equivalence to another type representation. `Display` produces a string representation of a type; `ScanStructure` and `FieldType` allow the components of a structure type to be enumerated; `Elms` returns the element type of a vector type; and `Args` and `Result` return the argument and result types of a procedure type.

Other procedures are supplied for manipulating variant and polymorphic types.

It is also possible, subject to certain security restrictions, to obtain the source code of a given procedure value using the following library procedure:

```
getProcSource : proc( any -> HyperSource )
```

where type `HyperSource` is defined as follows:

```
type HyperSource is structure(
     characters : string ;
     links :      List[ Substitution ] )
type Substitution is structure(
     value : any ; start, finish : int )
```

This allows the full closure of a procedure to be obtained in the form of a hyper-program representation, which contains both the text and links to the values of any free variables. Hyper-programs are discussed in more detail in [21].

3.2 Dynamic Compilation

There are a number of aspects to dynamic compilation: the mechanism for compilation itself, the granularity i.e. what is the unit of compilation with respect to the language, and how the environment of execution relates to the environment in which the program fragment is compiled. These will now be described in turn.

3.2.1 Compilation Mechanism

Several interfaces to the Napier88 compiler are available as procedures in the persistent store; the principal ones allowing compilation of strings and hyper-program representations are:

```
compileString :      proc( string ->
                           CompilationResult )
compileHyperSource : proc( HyperSource ->
                           CompilationResult )
```

where the result of an attempted compilation is an instance of the type:

```
type CompilationResult is variant(
     voidResult :    proc();
     nonVoidResult : proc( -> any );
     error :         List[ CompilationError ] )
```

The successful compilation of a void sequence, i.e. a program fragment which does not return a result, produces a result-less procedure whose body contains the compiled code. A non-void program fragment produces a procedure which will return the result as an instance of the infinite union type `any`, which can then be projected onto a specific type for use.

3.2.2 Compilation Granularity

The granularity of dynamic compilation in Napier88 is the sequence, which denotes a series of statements with or without a final result value. The result value may be any Napier88 value, including procedures.

```
let graham = Person( "Graham Kirby", 30 )
let sourceText = "proc( -> int )
                 begin
                      obj( age ) )
                 end"
let subs = Substitution( any( graham ), 27, 29 )
let hp = HyperSource( sourceText,
                      List[Substitution]( subs ) )
let res = compileHyperSource( hp )
```

Figure 3: Compiling a hyper-program source fragment

3.2.3 Compilation Environment

Napier88 program fragments are compiled independently of the environment in which they are generated. Since structural type equivalence is used [9], types defined within a generated fragment may match existing types in the persistent store.

During application evolution the generator may produce new data structures to be accessed by the newly generated code. At some stage these data structures need to be bound into the generated code. One simple way to achieve this is for the generators to make the data structures reachable from the root of the persistent store, from where they can be retrieved by the generated code. However, a potential problem then arises if other applications are active during the evolution process: unless the evolution takes place within an atomic transaction it is possible for some other application to interfere with the data structures before they are retrieved.

An alternative which does not rely on transactions is to bind the required data structures directly into the generated code as it is produced, giving a hyper-program representation. No retrieval from the persistent store is then needed, and it is not possible for other applications to adversely affect the evolution process. This is supported by the `compileHyperSource` interface to the compiler. Figure 3 shows an example of the construction and compilation of a hyper-program source fragment. This first creates a value of the structure type `Person`. It then declares a string `sourceText` containing the source code of a procedure which will return the person's current age, using a placeholder `obj` to denote the person value. The structure value `subs` contains the value itself and the position of the placeholder within the source text. The source is then combined with a substitution list to form the hyper-program representation `hp` which is compiled using `compileHyperSource`. During this the placeholder `obj` will be replaced by the reference to to `graham`.

3.3 Incremental Loading

The result of a successful dynamic compilation is a procedure value whose body contains the compiled code, thus the code is automatically loaded into the running program. Where no result is returned, the type of the procedure is `proc()` and it can be called without further checking. Compilation of a fragment with a return value gives a procedure of type `proc( -> any )` which returns the result as an instance of type `any`. A dynamic type check is then required to project the result onto the concrete type expected by the program using the compiled fragment. Figure 4 shows how the compiled result obtained in Figure 3 could be used.

```
project res as generatedCode onto
nonVoidResult :
begin
     let fragmentResult = generatedCode()
     project fragmentResult as getAge onto
     proc( -> int ) :
     begin
          let grahamsAge = getAge()
     end
     default : error( "compiled code
                    has unexpected result type" )
end
default : error( "compilation failed" )
```

Figure 4: Using dynamically compiled code

3.4 Example of Evolutionary Change

The example in this section, taken from [10], shows how an evolutionary change could be made to a simple Napier88 application. Although a trivial example, it serves to illustrate the mechanisms described earlier. We ignore many of the issues of schema evolution which are addressed in, for example, [16, 19, 23].

The schema of a very simple database application is shown below. The database contains information about a number of engineering parts, and the suppliers from which they may be obtained. This information is stored in the two persistent sets `PARTS` and `SUPPLIERS`, each of which contains instances of the types `Part` and `Supplier` respectively:

```
PARTS :     Set[Part]
SUPPLIERS : Set[Supplier]

type Address is ...

type Part is structure(
   p_name :    string;
   p_no :      int;
   suppliers : Set[Supplier] )

type Supplier is structure(
   s_name :    string;
   s_address : Address)
```

The procedure `nearest_supplier` implements a query which, given the factory address and the name of a part, returns the address of the nearest supplier of that part. It first finds the part by selecting those members of the `PARTS` set which match the given name, giving a set with a single element, and then picking out the element. It then calls the procedure `project_address` to project the address attribute from the set of suppliers of that part, giving a set of addresses, and finally calls another procedure `nearest_ad` to determine which address in that set is closest to the factory address.

```
! Returns the address of the nearest
! supplier of the given part.
let nearest_supplier =
  proc(ad : Address; name : string -> Address)
begin
  let filter = proc(p : Part -> bool)
               p(p_name) = name
  let this_part = pick[Part](
    select[Part](PARTS, filter))
  let ads = project_address(this_part(suppliers))
  nearest_ad(ad, ads)
end

! Returns member of ads which is closest to ad.
let nearest_ad = proc(ad : Address;
  ads : Set[Address] -> Address)
{...}
```

Thus the database as a whole comprises a set of type definitions describing the structure of the data, a set of persistent entry points, and a set of procedures which operate on the data. These procedures may be thought of as precompiled queries; ad-hoc queries can be supported by allowing users to enter new procedure definitions which are compiled and executed immediately.

There are a number of ways in which such a schema may evolve, the simplest being the addition or deletion of type attributes. Slightly more complex is the situation where the information being represented remains unchanged but it is modelled in a different way, for example by moving an attribute from one type to another. In all cases it is necessary to locate all affected type definitions, instances and queries (procedures), and to update them consistently in an atomic update to the database.

In Napier88 the first part is achieved by maintaining persistent links from each type definition to all the other type definitions, entry points and procedures which use it. It is thus relatively straightforward to locate the components affected by a given type change. Updated versions are then created and bound into the database using the mechanisms described earlier: source representations of the new versions are generated, compiled and loaded dynamically. For some simple varieties of schema change the generation of updated source representations can be automated; more generally user intervention is required to define precisely how the existing database values and procedures should be modified.

In the parts/suppliers example, a schema change might reverse the mapping from parts to suppliers, so that each representation of a supplier now contains a set of supplied parts. This is an example of a reorganisation which remodels the data without adding or removing information. The string shown below partially represents the new type definitions; it could be entered directly by the database administrator or generated by a schema change tool. The string is combined with a reference to the unchanged type `Address` to form a hyper-program representation of the new types, which is then compiled and the resulting types made available for constructing the new versions of the other database components.

```
"type Part is structure(
    p_name : string;
    p_no :   int)

 type Supplier is structure(
    s_name :         string;
    s_address :      Address;
    supplied_parts : Set[Part] )"
```

At some stage it will be necessary to update the data to conform with the new schema. This may be done eagerly for all data, or delayed until the data is accessed—or perhaps until a quiescent period. Whenever the conversion is performed, a conversion procedure is generated for each evolved type and then applied to each existing instance of

that type. The procedure produces a corresponding value conforming to the evolved schema. The source representation of a procedure to convert Supplier instances is shown below. Here Supplier, Part and PARTS denote links to the old versions while Supplier2 denotes the new version of the type. The body of this procedure is supplied by the database administrator (DBA): it specifies that each new supplier value contains the old name and address together with the set of parts which referred to that supplier in the old database. The procedure getNew takes a set of parts under the old schema and returns the set of corresponding new parts.

```
"proc(s : Supplier -> Supplier2)
 begin
   let filter = proc(p : Part -> bool)
     contains[Supplier](p(suppliers), s)
   let old_parts = select[Part](PARTS, filter)})
   let new_parts = getNew( old_parts )
   Supplier2(s(s_name),s(s_address),new_parts)
 end"
```

The final step is to update the stored queries which access the database. Again, a new string is generated from the source code obtained from the old version of each procedure. Here it is necessary for the DBA to specify that each occurrence of the pattern

```
X( suppliers )
```

in a procedure should be replaced by the pattern

```
let filter = proc(s : Supplier -> bool)
          contains[Part](s(supplied_parts), X)
select[Supplier](SUPPLIERS, filter)
```

Thus instead of accessing the suppliers attribute of a part, procedures must now scan all of the suppliers and select those whose parts set includes that part. The string representing the transformed nearest_supplier procedure is shown below. As before the string is combined with references to type definitions, entry points and other procedures to form the full hyper-program representation which is then compiled and loaded into the transformed database.

```
"proc(ad : Address; name : string -> Address)
 begin
   let filter = proc(p : Part -> bool)
                  p(p_name) = name
   let this_part = pick[Part](
     select[Part](PARTS, filter))
   let ads = project_address(
    { let filter = proc(s : Supplier -> bool)
      contains[Part](
        s(supplied_parts), this_part)
      select[Supplier](SUPPLIERS, filter)})
   nearest_ad(ad, ads)
 end"
```

To summarise, this example has illustrated how the types, data and procedures in a simple Napier88 database application may be evolved to a conform to a revised schema. In each case a source representation of the new version is generated, either automatically or through user intervention, and dynamically compiled and loaded into the evolved database.

4 Application Evolution in O_2

O_2 is an object-oriented database system with an explicit multiple inheritance model and persistence by reachability [12]. A database may be accessed by programs written in C, C++ or O_2C, which is a superset of C providing support for the database model. An object based query language OQL is also provided.

O_2 has some built-in support for schema evolution [13] which avoids the need for explicit application of transformation procedures to affected objects. The evolution of a persistent application involves the following steps:

- Meta-level descriptions of existing objects and classes are obtained using a meta-schema which is available as a predefined schema. It defines, among others, the classes Meta, Meta_class and Meta_type.
- Based on analysis of the meta-level descriptions, program fragments containing arbitrary O_2C or OQL code are generated in the form of strings. These fragments may contain revised class definitions, transformation functions and schema evolution commands. They are then compiled and executed in the context of the current database by invoking the Meta class method command or the function o2query.

These steps are now described in greater detail.

4.1 Meta-level Descriptions

Meta-level descriptions are provided in O_2C through the classes Meta, Meta_definition and Meta_class, part of which are shown in Figure 5. Together with the related classes Meta_type, Meta_collection and Meta_tuple they provide a complete meta-level description of the current schema. The Meta_class instance for a given object may be obtained by calling the class method of a Meta instance (which describes an entire schema).

A complete description of the current schema may be generated by invoking the Meta methods classes, types, names and functions. Further details can then be discovered by iterating over the structure of each list member obtained.

4.2 Dynamic Compilation

4.2.1 Compilation Mechanism

The ability to compile and execute program fragments is provided by the command method in the class Meta, and by the predefined function o2query. The former allows an arbitrary sequence of O_2C commands to be executed in the context of the current schema—these could be meta-commands which alter the schema, or commands to invoke particular methods. The o2query function is used to execute a query in OQL, the object query language, against the schema and return a result to an O_2C program. If an invalid program fragment is passed to command or o2query a non-zero value is returned.

4.2.2 Compilation Granularity

The granularity of compilation is the O_2C command in the case of command, and the OQL command in the case of o2query.

```
class Meta
  method
    public classes: list (
      tuple (name: string, class: Meta_class))
    public types: list (
      tuple (name: string, type: Meta_type))
    public names: list (
      tuple (name: string, def: Meta_definition))
    public functions: list (
      tuple (name: string, type: Meta_definition,
             parameters: list (
               tuple (name: string,
               type: Meta_definition))))
    public definition (name: string):
      Meta_definition
    public class (ob: Object): Meta_class
    public type (type_of_data: integer):
      Meta_definition
    public command (o2_command: string):
      integer
    ...
end

class Meta_definition
  Schema: Meta
  method
    public is_class: boolean
    public name: string
    ...
end

class Meta_class inherit Meta_definition
  method
    public attributes: list (
      tuple (name: string, visibility: string,
             type: Meta_definition))
    public methods: list (
      tuple (
         name: string, visibility: string,
         type: Meta_definition,
         parameters: list (
           tuple (name: string,
                  type: Meta_definition))))
    public type: Meta_definition
    public superclasses: list (Meta_class)
    public subclasses: list (Meta_class)
    ...
end
```

Figure 5: Meta classes

```
o2 int limit;
o2 set (int) low_salaries;
o2 set (Employee) employees;

/* initialise employees and limit */

o2query (low_salaries, "select e->salary
         from e in $1 where e->salary < $2",
         employees, limit);
```

Figure 6: Example of OQL query

4.2.3 Compilation Environment

O_2C commands and OQL queries are compiled and executed in the context of the current schema. In a similar manner to the Napier88 `compileHyperSource` procedure, the `o2query` function allows generated OQL queries to refer directly to objects in scope in the generating program, by means of placeholders. During execution of the query each occurrence of a query variable, of the form `$n`, within the OQL string is replaced by a corresponding actual parameter. In the example shown in Figure 6, the variable `low_salaries` is set to the result of a query which retrieves those salaries of a set of employees that fall below the threshold limit.

4.3 Incremental Loading

No explicit incremental loading mechanism is needed in O_2; the code passed to `command` or `o2query` is executed directly. The executable form of the code is not represented within the language, thus the programmer is not concerned with loading it into the running system.

4.4 Example of Evolutionary Change

This section shows how the example of Section 3.4 translates to O_2C. The data is stored in the persistent named sets `PARTS` and `SUPPLIERS`, containing instances of the classes `Part` and `Supplier`:

```
name PARTS:     set(Part);
name SUPPLIERS: set(Supplier);

class Part
public type tuple (p_name :    string,
                   p_no :      integer,
                   suppliers : set(Supplier))
end;

class Supplier
public type tuple (s_name :    string,
                   s_address : Address)
end;
```

The function `nearest_supplier` operates in the same way as in the Napier88 example. First it finds the part with the given name, then obtains the addresses of the part's suppliers, and finally calls the function `nearest_ad` to get the nearest address.

```
function body nearest_supplier (
  ad: Address, name: string) : Address
{
  o2 Part this_part;
```

```
  o2 set(Address) ads;

  for (p in PARTS where p.p_name == name)
    {this_part = p};
  ads = project_address (this_part.suppliers);
  return nearest_ad (ad, ads);
}
```

Evolving the application so that suppliers refer to parts requires the generation of several schema modification commands as strings, which are then passed to the command method. Again, the strings may be entered directly by the database administrator, or generated by a schema evolution tool. The first schema modification adds the new attribute supplied_parts to Supplier, and defines a conversion function gather_parts which will be used by the system to convert the existing instances of Supplier to the new format. The O_2 system guarantees that the conversion function will be applied to each instance at some time before it is accessed. When the function is called it is passed the old instance as a tuple; it then scans the parts set and inserts each part that refers to it into its own supplied_parts set.

```
"begin modification in class Supplier;
 create attribute supplied_parts : set(Part);
 conversion functions;
 conversion function gather_parts (old :
   tuple (s_name :    string,
          s_address : Address)) in class Supplier
 {
   for (p in PARTS where (self in p.suppliers))
     self->supplied_parts += p;
 };
 end modification;"
```

The second schema modification command deletes the now redundant attribute suppliers from class Part. Here no conversion function need be specified since a default conversion ensures that the attribute values become inaccessible. The O_2 system ensures however that they are not physically deleted before they are required by the conversion function specified in the previous schema modification.

```
"begin modification in class Part;
 delete attribute suppliers;
 end modification;"
```

Finally the nearest_supplier function is revised appropriately. This is achieved by executing a command which redefines the body of the function; as this is a relatively complex change to the code it probably requires intervention by the DBA.

```
"function body nearest_supplier (
   ad: Address, name: string) : Address
{
  o2 Part this_part;
  o2 set(Address) ads;
  o2 set(Supplier) sups;

  for (p in PARTS where p.p_name == name)
    {this_part = p};
  for (s in SUPPLIERS where
       this_part in s.supplied_parts)
    sups += s;
  ads = project_address (sups);
  return nearest_ad (ad, ads);
}"
```

To summarise, this section has illustrated how the schema of a simple O_2C application may be evolved. The built-in support for schema evolution means that the programmer need only specify how existing application data is to be modified, without considering explicitly when the changes are performed.

5 Application Evolution in Java

Java is an object-oriented language with an explicit single inheritance model and automatic memory management [15]. At the time of writing several research groups are developing persistent variants of Java with the aim of improving support for long-lived and data-intensive applications [4, 14]. The evolution of a persistent application in Java involves the following steps:

- Meta-level descriptions of existing objects and classes are obtained using a core reflection package which is part of a standard library. It defines, among others, the classes Class, Field and Method.
- Based on analysis of the meta-level descriptions, program fragments defining new classes are generated in the form of strings. These are then compiled by invoking the Java compiler to produce byte code sequences which comprise new class definitions.
- The dynamically created class definitions are then loaded into the executing system where they may be instantiated as new objects.

These steps are now described in greater detail.

5.1 Meta-level Descriptions

Meta-level descriptions are provided in Java through the class Class, a unique instance of which is associated with each Java class. The Class instance for a given object may be obtained by calling the following method which is inherited by every object from the root class Object:

```
public final Class getClass();
```

The class Class and the related classes Array, Constructor, Field, Method and Modifier provide complete meta-level descriptions of the class hierarchy. Part of the definition of Class, that relevant to the discussion, is shown in Figure 7; details of exceptions thrown have been omitted to save space.

A complete description of the class of an object may be generated by invoking the getClass method followed by invoking the various Class methods to obtain representations of the class's fields, methods, constructors, superclass and interfaces. These representations in turn provide methods which allow their structure, such as the types of fields and the parameter types of methods, to be discovered. Where such components are object types the process can be continued recursively.

5.2 Dynamic Compilation

5.2.1 Compilation Mechanism

Although no facilities for dynamic compilation are provided directly by the Java environment, it is possible to implement a class which provides such facilities. Figure 8 shows an example providing a compile method which takes a source

```
public final class Class {
  public static Class forName( String className );
  public Object newInstance();
  public String getName();
  public Class getSuperclass();
  public Class[] getInterfaces();
  public int getModifiers();
  public Class[] getDeclaredClasses();
  public Field[] getDeclaredFields();
  public Method[] getDeclaredMethods();
  public Constructor[] getDeclaredConstructors();
  public Field getDeclaredField( String name );
  public Method getDeclaredMethod(
     String name, Class parameterTypes[]);
  public Constructor getDeclaredConstructor(
     Class parameterTypes[] );
  ...
```

Figure 7: The class *Class*

```
public class Compiler {
     // A run-time callable compiler.
     public Class[] compile( String source )
     throws CompilationError { ... }
     ...
}
```

Figure 8: Outline of a compiler class

code string defining one or more classes, and attempts to compile it to an array of instances of class `Class`.

Such a method may be implemented by splitting the source string into separate class definitions and then writing these strings into files. The standard Java compiler is then invoked in order to compile the source files into *.class* byte-code files. There are several techniques which may be used to invoke the compiler from a running program; these are omitted here due to space constraints but they are described in [17]. Finally the compiled code is loaded back into the running system using a customised class loader—this is discussed in Section 5.3.

5.2.2 Compilation Granularity

The granularity of dynamic compilation in Java is the class: a string to be compiled must contain one or more complete class definitions. This is dictated by the facts that the standard Java compiler operates on source files containing complete class definitions, and that the incremental loading mechanism to be described operates on classes.

5.2.3 Compilation Environment

A dynamically compiled program fragment may use existing classes in two different ways. One is for the fragment to contain an **import** clause specifying a number of classes or packages to be brought into scope. Their use within the program fragment is checked by the compiler against their compiled definitions which are found by searching the directories in the classpath. On execution of the fragment, i.e. instantiation of a dynamically created class, any existing classes which are used are loaded as necessary by the run-time system.

```
public abstract class ClassLoader {
  public Class loadClass( String name );
  protected abstract Class loadClass(
     String name, boolean resolve );
  protected final Class defineClass(
     byte data[], int offset, int length );
  protected final void resolveClass( Class c );
  protected final Class findSystemClass(
     String name );
  ...
}
```

Figure 9: The class *ClassLoader*

Another possibility is for the program fragment to use the static method `forName` of class `Class` to obtain a class object from a class name in string form. Again the class definition is located in a classpath directory by the run-time system. It is not possible for a generated fragment to refer directly to an object which is in scope in the generating code. With a persistent Java indirect access may be achieved if the generating code makes the object persistent and the generated fragment includes code to retrieve the object from the persistent store. However there is no guarantee that object will remain accessible for the lifetime of the generated code.

5.3 Incremental Loading

Dynamic compilation creates class definitions in the form of byte code sequences. To be useful these must then be loaded into the running system and converted to `Class` objects. This may be achieved by defining and using a subclass of the class `ClassLoader`, which is partially defined in Figure 9.

Note that the methods `defineClass`, `resolveClass` and `findSystemClass` are **final** and therefore may not be overridden in a subclass, while `loadClass (String, boolean)` is **abstract** and must be overridden. Each subclass of the class `ClassLoader` provides an implementation of `loadClass` which maps class names to the corresponding byte sequences in a particular way. It might for example search a file system directory for an appropriate *.class* file, in a similar fashion to the run-time system's behaviour, or it might read bytes directly from a byte array without accessing the file system.

Once a class has been loaded at run-time an instance of the class can be created as illustrated in Figures 10 and 11. This assumes the definition of a subclass of `ClassLoader` named `DirSpecificClassLoader` which loads classes from a directory specified at its creation. Note that although an instance of any class can be created, it is initially treated as an instance of class `Object`. If any more specific methods are to be invoked their signatures must be known statically and the instance cast to a class or interface defining them.

The use of class loaders provides the final step in the dynamic creation and use of new code.

5.4 Example of Evolutionary Change

This section shows how the example of Section 3.4 translates to Java. The data is stored in the persistent hashtables `PARTS` and `SUPPLIERS` which map part and supplier names to the corresponding objects of the classes `Part` and `Supplier`:

```
File classDir = new File( "/user/gk/classes" );
DirSpecificClassLoader loader =
    new DirSpecificClassLoader( classDir );
try {
    Class newClass = loader.loadClass( "Person" );
    Object aPerson = newClass.newInstance();

    // Assumes class Person was defined
    // to implement PersonInterface.
    PersonInterface usablePerson =
          (PersonInterface) aPerson;
    usablePerson.writeAge();
} catch ( Exception e ) {
    System.out.println( "class loading error" );
}
```

Figure 10: Using a dynamically loaded class

```
interface PersonInterface {
     public void writeAge();
```

Figure 11: A statically known interface

```
Hashtable PARTS;
Hashtable SUPPLIERS;

public class Part {
  public String    p_name;
  public int       p_no;
  public HashTable suppliers;
}

public class Supplier {
  public String  s_name;
  public Address s_address;
}
```

The method nearest_supplier operates in the same way as in the previous examples. First it looks up the part name in the PARTS hashtable and casts the object to class Part, then it obtains the addresses of the part's suppliers, and finally calls the method nearest_ad to get the nearest address.

```
public static Address nearest_supplier (
  Address ad, String name)
{
  Part this_part = (Part) PARTS.get (name);
  Hashtable ads = project_address (
    this_part.suppliers);
  return nearest_ad (ad, ads);
}
```

Evolving the application so that suppliers refer to parts requires the generation of strings containing revised class definitions. New class names must be used since there are no schema evolution facilities built into Java, and class equivalence is by matching on class names. The strings are compiled dynamically and the resulting class definitions loaded to give the evolved classes:

```
"public class Part2 {
   public String p_name;
   public int    p_no;
 }"

"public class Supplier2 {
   public String    s_name;
   public Address   s_address;
   public HashTable supplied_parts;
 }"
```

The existing part and supplier objects must then be converted to the new classes. This is achieved by scanning the persistent hashtables, converting each object and entering the resulting object in a corresponding new hashtable. The code to perform the conversion is generated as a string defining a class which is compiled, loaded and the main method called. For example, to convert the existing supplier objects:

```
"public class ConvertSuppliers{
   static void main() {
     Enumeration suppliers = SUPPLIERS.elements();
     while (suppliers.hasMoreElements()) {
       Supplier s =
         (Supplier) suppliers.nextElement();
       HashTable pts = new Hashtable();
       Enumeration parts = PARTS.elements();
       while (parts.hasMoreElements()) {
         Part p = (Part) parts.nextElement();
         if (p.suppliers.get (s.s_name) != null)
           pts.put (p.p_name, p);
       }
       Hashtable new_parts = getNew (pts);
       Supplier2 converted = new Supplier2 (
         s.s_name, s.s_address, new_parts);
       SUPPLIERS2.put (
         converted.s_name, converted);
     }
   }
 }"
```

The nearest_supplier method must also be revised. This can be achieved either by including it in the class Supplier2, or by generating the representation of some other new class which is then compiled and loaded:

```
"public static Address nearest_supplier (
   Address ad, String name)
 {
   HashTable sups = new Hashtable();
   Enumeration suppliers = SUPPLIERS2.elements();
   while (suppliers.hasMoreElements()) {
     Supplier2 s =
       (Supplier2) suppliers.nextElement();
     if (s.supplied_parts.get (name) != null)
       sups.put (s.s_name, s);
   }
   Hashtable ads = project_address (sups);
   return nearest_ad (ad, ads);
 }"
```

To summarise, this section has illustrated how the schema of a simple Java application may be evolved, by generating strings representing new classes which are dynamically compiled. The classes may then be loaded, new instances created, and static methods called. Since no built-in support for schema evolution is provided, it is up to the DBA to organise the migration of existing data to the new schema.

6 Conclusions

The message in this paper is that research ideas may be delivered to commercial systems. We have proposed a solution to the problem of mechanising system evolution in persistent/database systems. We note that the problem is intractable in general but where it is understood it can be mechanised in such a fashion that the application itself or a specific evolution tool can perform the evolution. Such a mechanism lays the foundation for self evolving applications and systems. The mechanism proposed for system evolution is taken from the research language Napier88 and involves the capturing of meta-information, dynamic compilation and incremental loading. It is sometimes referred to as type safe linguistic reflection. We have shown how the technique may be used in two representative commercial systems, namely Java with a persistent store and O_2C.

References

[1] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In R. Agrawal, S. Baker, and D. Bell, editors, *19th International Conference on Very Large Data Bases*, pages 39–51, Dublin, Ireland, 1993. Morgan Kaufmann.

[2] A. Albano, L. Cardelli, and R. Orsini. Galileo: a strongly typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[3] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, 1983.

[4] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *SIGMOD Record*, 25(4):68–75, 1996.

[5] M. P. Atkinson, M. J. Jordan, L. Daynès, and S. Spence. Design issues for persistent Java: a type-safe, object-oriented, orthogonally persistent system. In R. C. H. Connor and S. Nettles, editors, *7th International Workshop on Persistent Object Systems*, pages 33–47, Cape May, NJ, USA, 1996. Morgan Kaufmann.

[6] M. P. Atkinson and R. Morrison. Orthogonally persistent object systems. *VLDB Journal*, 4(3):319–401, 1995.

[7] L. Cardelli. Amber. In *Lecture Notes in Computer Science 242*, pages 21–47. Springer-Verlag, Berlin, 1986.

[8] R. C. H. Connor. *Types and Polymorphism in Persistent Programming Systems.* Ph.D., University of St Andrews, 1990.

[9] R. C. H. Connor, A. B. Brown, Q. I. Cutts, A. Dearle, R. Morrison, and J. Rosenberg. Type equivalence checking in persistent object systems. In A. Dearle, G. M. Shaw, and S. B. Zdonik, editors, *Implementing Persistent Object Bases, Principles and Practice*, pages 151–164. Morgan Kaufmann, 1990.

[10] R. C. H. Connor, Q. I. Cutts, G. N. C. Kirby, and R. Morrison. Using persistence technology to control schema evolution. In E. Deaton, D. Oppenheim, J. Urban, and H. Berghel, editors, *9th ACM Symposium on Applied Computing*, pages 441–446, Phoenix, Arizona, 1994. ACM Press.

[11] I. F. Currie. Filestore and modes in Flex. In M. P. Atkinson, O. P. Buneman, and R. Morrison, editors, *1st International Workshop on Persistent Object Systems*, pages 325–334, Appin, Scotland, 1985.

[12] O. Deux. The O_2 system. *Communications of the ACM*, 34(10):34–48, 1991.

[13] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and database evolution in the O_2 object database system. In U. Dayal, P.M.D. Gray, and S. Nishio, editors, *21st International Conference on Very Large Data Bases*, pages 170–181, Zürich, Switzerland, 1995. Morgan Kaufmann.

[14] A. Garthwaite and S. Nettles. Transactions for Java. In *1st International Workshop on Persistence for Java*, Glasgow, 1996.

[15] J. Gosling and H. McGilton. The Java language environment: A white paper. Technical Report, Sun Microsystems, Inc, 1995.

[16] B. S. Lerner and A. N. Habermann. Beyond schema evolution to database reorganization. *SIGPLAN Notices*, 25(10):67–76, 1990.

[17] G. N. C. Kirby, R. Morrison, R. C. H. Connor, and D. W. Stemple. Linguistic reflection as a paradigm for program generation in Java. Submitted for publication, 1997.

[18] F. Matthes and S. Müßig. The Tycoon Language TL: An introduction. Technical Report DBIS 112-93, University of Hamburg, Germany, 1993.

[19] S. R. Monk. *A Model for Schema Evolution in Object-Oriented Database Systems.* Ph.D., Lancaster University, 1993.

[20] R. Morrison, A. L. Brown, R. C. H. Connor, Q. I. Cutts, A. Dearle, G. N. C. Kirby, and D. S. Munro. Napier88 reference manual (release 2.2.1). Technical Report, University of St Andrews, 1996. Release 2.0 version published as CS/94/8.

[21] R. Morrison, R. C. H. Connor, Q. I. Cutts, V. S. Dunstan, and G. N. C. Kirby. Exploiting persistent linkage in software engineering environments. *Computer Journal*, 38(1):1–16, 1995.

[22] PS-algol reference manual, 4th edition. Technical Report PPRR-12-88, Universities of Glasgow and St Andrews, 1988.

[23] J. F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.

[24] C. Schaffert, T. Cooper, and C. Wilpot. Trellis object-based environment language reference manual. Technical Report 372, DEC Systems Research Center, 1985.

[25] D. Stemple, R. B. Stanton, T. Sheard, P. Philbrow, R. Morrison, G. N. C. Kirby, L. Fegaras, R. L. Cooper, R. C. H. Connor, M. P. Atkinson, and S. Alagic. Type-safe linguistic reflection: a generator technology. Technical Report FIDE/92/49, ESPRIT BRA Project 3070 FIDE, 1992.