Software Components for Computer Algebra

Pietro Iglio
via Nasini 12
I-00156 Roma, Italy
iglio@fub.it

Giuseppe Attardi
Dipartimento di Informatica
corso Italia 40, I-56125 Pisa, Italy
attardi@di.unipi.it

Abstract

Software components encourage code reuse and simplify application development. An increasing number of applications is built assembling components developed by third parties, taking advantage of language-independence, object orientation, ease of use and other features of modern component architectures.

Computer algebra systems could exploit the software component approach, but several issues must be addressed, mostly due to the sophisticated data structures required for representing mathematical objects. We discuss these problems and present a proposal based on the OpenMath specifications.

We built an prototype framework for developing and using mathematical components. The framework uses IDL from CORBA for specifying the interfaces for objects. Code developed in the framework is mapped into either the COM object model for creating ActiveX components or into CORBA objects for creating servers implemented as dynamic modules.

1. Introduction

A major trend in the evolution of software technology has been to favour code reuse in writing new applications. Modular programming and object oriented programming provided only a partial solution to this problem.

Recently a new approach, centred on the concept of *software component*, has emerged as a successful technology for building modular and reusable code. Software components mimics the metaphor of electronic components: engineers design electronic devices by combining off-the-shelf components, whose behaviour and characteristics are precisely specified, and which have been tested separately, and connect their pins according to their specifications, without knowledge of their internal details.

From a technical point of view, a software component is similar to an object in an object oriented programming language. However a component is generally language-independent, is distributed in the form of a binary library and can be dynamically added to the component tool box of many modern development tools. To achieve this feature a software component has two separate interfaces: a *configuration interface* and a *programming interface*. The programming interface is used at run time by applications of the component. The configuration interface is used during application development by program development tools for inspecting a component, i.e. determining which methods, properties and events it exports, and allowing the user to select parameters for configuring the component in the application, for instance its initialisation parameters or its graphics properties. This ability to integrate software components with minimal effort is probably a major reason for their success. A component can be dealt with any of the several development tools which support component, so programmers are not bound to a specific programming environment nor to a specific programming language. A large number of third party components is already available and so building complex applications is often just a matter of selecting suitable components and assembling them with simple point-and-click operations.

The most prominent component architectures are currently JavaBeans [JVB] and Microsoft ActiveX/COM [BRO, DEN].

JavaBeans is the platform-neutral, component architecture for the Java language. Its popularity is partially due to the popularity of this language. The platform independence is achieved by means of an intermediate bytecode which is interpreted by the Java Virtual Machine, with a consequent loss of performance (however, modern Java platform address this problem with Just-In-Time compilers). JavaBeans is a recent technology, but is quite well designed. Several development tools already support JavaBeans, including visual environments which enable connecting components to build applications through a graphical interface.

ActiveX/COM is the component architecture of Microsoft. This architecture is in wide use and a large number of components and applications are based on this standard. However, it is limited to Windows platforms (even though recently Software AG has released a UNIX version [ENT]) and it is based on an object model which does not support inheritance.

Other component architectures are IBM SOM [SOM] and OpenDoc [OPE], both based on the CORBA specifications [COR]. CORBA (Common Object Request Broker Architecture), is a distributed object

computing infrastructure, defined by the Object Management Group. CORBA allows transparent access to objects distributed over a network, and provides a set of services and facilities to locate objects, to activate them remotely.

Despite its advantages, the software component paradigm has been rarely considered in the field of computer algebra. One example is the PolyMath library [JOL], a library written in Java which implements the basic OpenMath standard. The component model for PolyMath is, of course, JavaBeans. The work done for the PolyMath library is, in some respects, similar to the work presented here.

The purpose of this paper is to examine the problems related to the peculiarities of mathematical software and to propose an architecture for mathematical software components.

In the next section we introduce the concept of mathematical component, In section 3 we show how abstract interfaces can address the problem of data representation. In section 4 and 5 we describe how OpenMath objects can been specified using the OMG Interface Definition Language, while in section 6 we provide specifications for mathematical components. In section 7 we present a framework for developing mathematical components along the lines discussed earlier. In sections 8 and 9 we illustrate how the framework is mapped into an object model and present some tools to help programmers in this task.

2. Mathematical Components

Computer algebra systems could take advantage of a component architecture that supports mathematical components, i.e. software components representing mathematical objects and algorithms.

Consider the following problem: implementing a particular algorithm and making it available to the mathematical community. Since there are several computer algebra systems in use, one would have to code the algorithm in each of these systems, using their specific extension language. Alternatively one could use *sockets* to exchange input/output data between the computer algebra system and an external program implementing the algorithm.

Both approaches have drawbacks: in the first case one has to re-code the algorithm several times, one for each computer algebra systems to support; moreover, many extension languages are interpreted and so this solution does not provide the uttermost efficiency. In the second case, the socket-based approach, one must define a communication protocol for exchanging mathematical data and implement the client side of the protocol for each computer algebra system; moreover, there can be significant communication overheads: if the computation is simple or the data are large, the time spent to send and receive the mathematical structures can be longer than the computation time.

As far as the communication protocol is concerned, there are several attempts to standardise the representation of mathematical objects. One of them is the OpenMath Consortium [OMA], whose goal is to define a platform-independent standard for the representation of mathematical objects so that they may be exchanged in a meaningful way between various software tools. A diagram showing two programs communicating with each other using the OpenMath protocol is shown in figure 1.

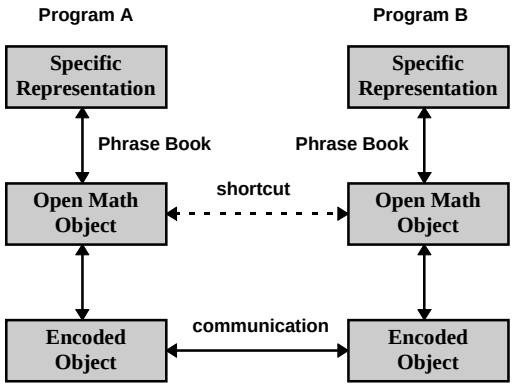

Figure 1: The OpenMath Architecture

A *Phrase Book* is software which translates the application specific representation into OpenMath objects and *vice versa*. The OpenMath objects are recursive data structures describing mathematical objects. Their formal definition is given by one or more *Content Dictionaries* (CDs). The OpenMath objects are encoded in byte streams. OpenMath compliant implementations must support SGML encoding. An example of SGML encoding for the mathematical expression *sin*(*x*+*y*) is: `<sin> <plus> x y </plus> </sin>`.

Our approach is based on the OpenMath proposal, and can be considered as a particular implementation of it. Most current implementation of OpenMath, in fact, employ the socket-based approach, while our proposal defines a plug-in interface so that binary components can be dynamically loaded into any compliant computer algebra system, providing an extension to its kernel.

The advantages of a similar solution are:

- a market for third party mathematical components could be established, since a single component could target several computer algebra systems at once;
- certain algorithms within computer algebra systems could be replaced with improved versions;
- mathematical components can be used directly within any computing environment, not necessarily a computer algebra one.

The dynamic loading of a software module is not itself a problem and is already possible with some systems. For instance, the MuPAD system [MUP] already supports a similar mechanisms, called *dynamic modules* [SOR] and based on the concept of shared

library. A dynamic module must be written in C or C++, according to some specifications, then compiled and linked into a shared library by means of a specific tool. Such module can be loaded and unloaded into the MuPAD environment as needed, and it becomes like a kernel extension.

However, such modules cannot be used by other computer algebra systems, since the procedures within a dynamic module directly access the native data representation of the MuPAD system. For instance a dynamic module for polynomials must be programmed either to use the MuPAD representation for polynomials or to convert them into its own format. In both cases, one cannot use the same component with another computer algebra system.

Therefore, data representation and conversion for mathematical objects is a major issue for building system-independent components.

3. Separating Interfaces from Implementations

Mathematical structures can be implemented in a number of ways. If a computer algebra system represents a polynomial as lists of monomials and a component represents them as vectors of monomials, how can the computer algebra system pass polynomials to the component and interpret the result of the computation?

To address the problem we resort to one of the basic tenets of object oriented programming: separating interfaces from implementations. Interfaces provide methods to access the mathematical structures in an abstract way, without any assumption on the underlying implementation. Considering the previous example, if the client (e.g. the computer algebra system) provides access to its mathematical objects through an abstract interface, the server (the component) can access the object through that interface, for instance it can access the n-*th* monomial of a polynomial without knowing whether it is represented as a list or as a vector.

The C++ language provides a mechanism for separating interfaces and implementations by means of abstract classes. The above example could be written as follows:

```
// Abstract Interface for Polynomials
class IPoly {
     virtual IMonom& getMonomial(int n) = 0;
     ...
};

// First implementation
class PolyAsList : public IPoly {
private:
     List<IMonom> monomials;

public:
     virtual IMonom& getMonomial(int n) {
            return monomials.getNth(n);
     }
}
// Second implementation
class PolyAsVector : public IPoly {
private:
     Vector<IMonom> monomials;

public:
     virtual IMonom& getMonomial(int n) {
            return monomials[n];
     }
}
```

Using a common abstract interface, the computer algebra system and the component could share polynomials. Both will use, for example:

```
poly->getMonomial(0)
```

to access the first monomial, irrespectively of whether the polynomial is represented as a list or as vector.

4. Object Interface Specifications

Starting from the OpenMath specifications, we wrote interfaces for most of the objects in the Basic and Poly CDs (in the rest of the paper we will refer to them as OM objects), using the Interface Definition Language (IDL) [COR], developed by the Object Management Group (OMG).

IDL is part of the CORBA architecture, introduced in section 1. Using IDL to define object interfaces has some potential advantages over using C++, in particular:

- IDL enforces a policy of separating interfaces and implementation;
- IDL can be mapped in a natural way to more than one programming language; binding for C++, C, Lisp, Java and other languages are currently available;
- IDL provides platform independent specifications for basic data types;
- from IDL code can be automatically generated for basic methods like serialization, stubs for remote procedure invocation, etc.

Furthermore, IDL supports inheritance and its syntax is very similar to the C++ class declaration syntax. The main differences are the use of the keyword **`interface`** instead of the keyword **`class`**, the absence of private and protected inheritance, and the specification of **`in`**, **`out`** or **`inout`** for parameter types in method declarations. Such attributes specify respectively a parameter passed from client to server, from server to client and in both directions.

IDL also defines a special template type: `sequence<T [, max_size]>`, representing a one-dimensional array of type `T`. If a maximum size is specified, then the sequence is a bounded sequence.

The IDL specifications for OM objects provides the basic operations to access, traverse, modify and copy objects.

In the definition of object interfaces we followed as much as possible the current OM definition. For

instance, since a OM distributed multivariate polynomial is defined as follows:

```
<CDDefinition>
  <Name> DMP </Name>
  <Description>
   The constructor of DMPs. The first argument is the
polynomial ring containing the polynomial and the second is a
"SDMP". Should be of the form DMP(PolyRing(...), SDMP(...))
  </Description>
  <FunctorClass> Constructor, Binary </FunctorClass>
</CDDefinition>
```

we have defined an interface `IDMP` (where "`I`" is the prefix for interfaces) as follows:

```
interface IDMP : IObj {
     IPolyRing getPolyRing();
     void setPolyRing(in IPolyRing pr);
     ISDMP getSDMP();
     void setSDMP(in ISDMP sdmp);
};
```

For each object subcomponent, a pair of (`set`, `get`) methods has been specified to retrieve and assign the subcomponent, such as `getSDMP()`/`setSDMP()` in the previous example.

The root of the inheritance diagram for OM objects is the `IObj` interface, from which they inherit methods common to all OM objects. The main methods exported by `IObj` are:

`copy()`: performs a deep copy of the object; the copy is performed using a factory object supplied as argument (see next section), that will be used to create the new object. This allows creating a copy of a given object in an alternative implementation.

`serialize()`: generates an OpenMath SGML encoding that can be saved into a text file or sent to a remote OpenMath server. This way it is possible to get interaction between programs using implementation of our interfaces and stream-based servers;

`equals()`: checks structural equivalency between two objects. The result can be `true`, `false` or `dontKnow`. The `dontKnow` value is returned in case the equivalence between the two mathematical objects cannot be established;

`getAttributes()`, `setAttributes()`: to access/modify the sequence of attributes associated to the object (attributes are a requirement of the OpenMath standard);

All atomic objects, such as integers and strings, inherit directly from `IObj`, as well as any other type defined in additional CDs.

OpenMath needs also the ability to represent expressions, which may be passed between algebra systems for evaluation or to typesetting or graphics systems for display. Such expressions are represented by means of the class `OMExpr`, which has three subclasses `OMConstant`, `OMVariable`, `OMTerm`. The last one is used to represent functional expressions and provides the following interface:

```
    OMExpr getFunctor(in OMTerm);
    void setFunctor(in OMTerm, in OMExpr);
    OMExpr   getParameter(in   OMTerm,   in
long);
    void setParameter(in OMTerm, in long,
in OMExpr);
```

Since the current OM specifications does not provide a complex type hierarchy, we decided to keep the inheritance diagram as simple as possible to reflect that choice. Inheritance has been mostly used to simplify the implementation of interfaces, and not to detect type errors at compile-time. For instance, since there is no category for polynomial coefficients in the OpenMath specifications, constructors in our specifications allow building a monomial with any OM object as coefficient. Each implementation is responsible to perform run time checks to detect errors.

The OpenMath object architecture allows manipulating objects directly through their interface, without having to transform them into the generic OpenMath representation, as shown in Figure 2.

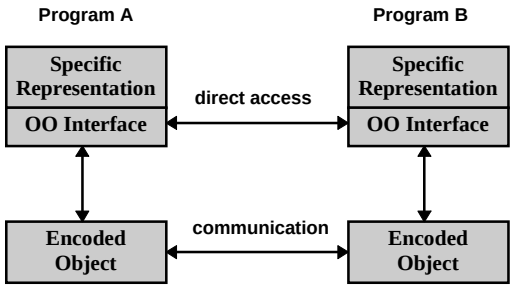

Figure 2: OpenMath Objects Interactions

5. Building OM Objects

Some special objects, called *factory objects*, provide methods to create other OM objects. Since the OpenMath specifications groups objects into *Content Dictionaries* (CD), our specifications associates factory objects to CDs. The `IBasicCD` interface provides methods to build OM objects belonging to the Basic CD, for instance:

```
IInteger createInteger(in long value);
```

while the `IPolyCD` interface provides methods to build OM objects belonging to the Poly CD, such as:

```
IMonom createMonomial(in IObj coeff,
                      in long nExponents);
ISDMP createSDMP(in long nMonomials);
```

Factory objects, provide a mean to select among different implementations at run time. For instance, the following function shows a possible use of the `IBasicCD` factory:

```
foo(IBasicCD* basicCD)
{
  IInteger* one =
     basicCD->createInteger(1);
  IInteger* two =
     basicCD->createInteger(2);

  OMExpr sum =
     basicCD->createTerm("plus", 2);
  sum->setParameter(one, 0);
  sum->setParameter(two, 1);

  cout << sum->serialize() << " = "
       << basicCD->plus(a, b);
}
```

Passing an `IBasicCD` factory to the above function will output:

```
<plus> 1 2 </plus> = 3
```

since `sum` will be an OM object corresponding to the expression `1 + 2`. Serialising an object returns its external representation, in our case an SGML representation.

By using factories, the framework can also provide a general conversion facility between object representations, via the method `copy()`. This method accepts as argument the factory for the target representation: it traverses recursively the object and reconstructs a copy of it using the supplied factory. For instance, given the factories for two implementations of the PolyCD, `PCD1` and `PCD2`, we can build a polynomial in the first one, convert it to the second and add it to another polynomial in the second implementation:

```
p1 = PCD1->createPoly("3x2+y");
p2 = PCD2->createPoly("5x+y3");
p3 = p1->copy(PCD2);
p3->add(p2);
```

6. Component Interface

Our framework provides also a configuration interface to the component, which is required for the packaging and distribution of mathematical components. Methods from these interfaces allow a computer algebra system to retrieve information about the component such as version, author, copyright. It is typical of development tools supporting software components to provide such inspecting feature. In Java most of the information for inspection is obtained through *introspection*.

We also provide a second interface which helps organising libraries which export a large number of functions.

The two interfaces for mathematical components are `IComponent` and `Ipackage`, as shown in figure 3.

`getContentDictionary()` returns the factory of a component. For instance, if `comp` is a reference to a `IComponent` object:

```
IBasicCD* basic =
   comp->getContentDictionary(OM_BASIC_CD);
```

```
IInteger* val = basic->createInteger(10);
```

will create an integer using the component implementation for multiple precision integers.

```
interface IComponent {
   IContentDictionary getContentDictionary
            (in OmContentDictionaryId cdId);

   IPackage getPackage(in string
                       interfaceName);
   IPackageSeq getPackages();

   void init();
   void finish();

   string getName();
   short getVersion();
short getInternalVersion();

   string getAuthor();
   string getCopyright();
   boolean getLicencing();

   string getURL();
};

interface IPackage {
   IObj invoke(in string method,
               in IObjSeq parameters);
   short getVersion();
short getInternalVersion();

   string getAuthor();
   string getCopyright();
   boolean getLicencing();
};
```

Figure 3: IComponent and IPackage interfaces

In a component a set of functions concerning a particular mathematical area can be grouped into packages. The method `getPackages()` returns the sequence of all packages included in the component. A client program can examine individually each package using the methods in `IPackage`. The method `getPackage()` retrieves a package given its name, e.g.:

```
IPackage* linalg =
   comp->getPackage("Linear Algebra");
```

The methods `init()` and `finish()` are used for component initialisation/finalisation, while `getName()`, `getVersion()`, `getAuthor()`, `getCopyright()` and `getLicencing()` are used to retrieve general information about the component. The method `getURL()` allows a client program to connect with a remote server (such as an Internet Web server) to download updated version of the component.

The only relevant method for the `IPackage` interface is `invoke()`, which allows dynamic invocation based on the function name passed as a string. This method is useful for invoking package functions from within scripting languages, such as those available with many computer algebra systems.

Each package will be describe by an interface derived from the interface `IPackage`, which lists the functions exported by a package. For example:

```
class IPackagePoly : public virtual IPackage
{
  // Implementation for IPackage methods
  string getName() {
     return "Polynomial Package";
  }
  ...
  // User methods
  ISDMP* algorithm1(IMonom* mono);
  IDMP* algorithm2(IInteger* val);
  ...
};
```

describes a package which includes implementations for methods `algorithm1()`, `algorithm2()`.

7. The O³ Framework

We built an experimental framework, called O³, which provides a C++ implementation of the above interfaces.

The framework consists in:

- a set of abstract implementations, i.e. classes which provide an implementation of some abstract interfaces in order to simplify the development of actual implementations;
- a concrete implementation of a subset of the OM interfaces.

The relationships between abstract interfaces, abstract implementations and implementations are shown in figure 4. Abstract interfaces are represented by C++ abstract classes. Abstract implementations provide a default implementation for some methods, reducing the code to be written for custom implementations. Abstract implementations have the "`Om`" suffix. The `OmObj` class is the abstract implementation for `IObj` and provides default implementation for methods such `copy()`, `serialize()`, `equals()`. `OmSDMP` is the concrete interface for the simple distributed multivariate polynomial and `OmSDMPa` and `OmSDMPb` are two possible different implementations of the same interface.

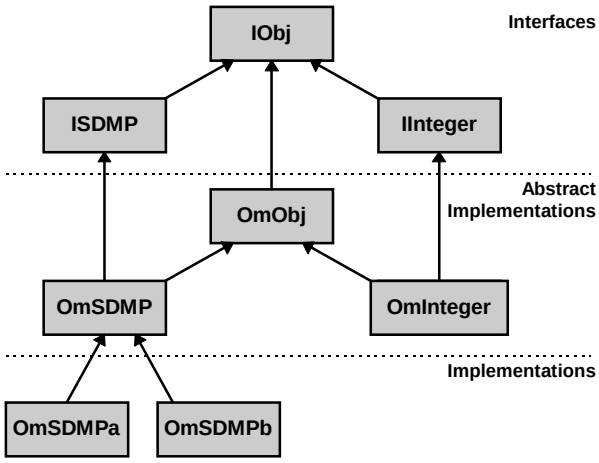

Figure 4: O³ Inheritance Diagram

The final result is a set of abstract classes from which a custom implementation can inherit, thereby reducing the amount of code to be written. For instance, by inheriting from class `OmMonom`, implementing interface `IMonom` requires just writing implementations for constructors `get/setCoefficient()` and methods `get/setExponent()`.

The concrete implementation includes the constructors for objects specified in the Basic and Poly CD, according to the OpenMath specifications version 1.2.

8. Mapping O³ to an object model

The elements of the O³ framework must be mapped to an actual object model. For lack of a satisfying component model encompassing both Windows and Unix, we decided to produce two mappings, one to COM through the ATL (ActiveX Template Library) for Windows and one we decided to produce two mappings, one to COM through the ATL (ActiveX Template Library) for Windows and one to CORBA for Unix, using shared libraries to create dynamic modules.

ATL [ATL] is a lightweight class library for building ActiveX components. To use ATL, we must translate IDL interfaces into Custom OLE Interfaces, which consists of C++ definitions plus additional annotations required for building the methods maps of the ActiveX component. This task can be done by a modified version of an IDL compiler. For instance, the concrete class `CMonomial` will contain:

```
BEGIN_COM_MAP(CMonomial)
    COM_INTERFACE_ENTRY(IMonomial)
END_COM_MAP()
```

Similarly, accessing the factory of a component, like PolyCD, is performed like this:

```
CoCreateInstance(CLSID_CPolyCD, NULL,
    CLSCTX_INPROC_SERVER,
    IID_IUnknown,(void**)&polyCD);
```

while creating a monomial with such factory involves:

```
polyCD->QueryInterface(IID_IMonomial,
                       (LPVOID*)&pMon);
```

This code can be obtained through preprocessing from a single set of IDL and C++ source. After preprocessing, this code can be given to the C++ compiler for Windows to produce an ActiveX software component, which can be registered and used in any component based application. For instance such component can be embedded into a spreadsheet and be used to perform symbolic computations whose input and output are exchanged through cells of the spreadsheet. We have carried out an experiment of this kind with the PoSSo library [POS], using it as a component to compute the Gröbner basis of polynomials entered in an Excel worksheet, with part of the output being displayed graphically in a chart, created by means of a chart component, as shown in Figure 5.

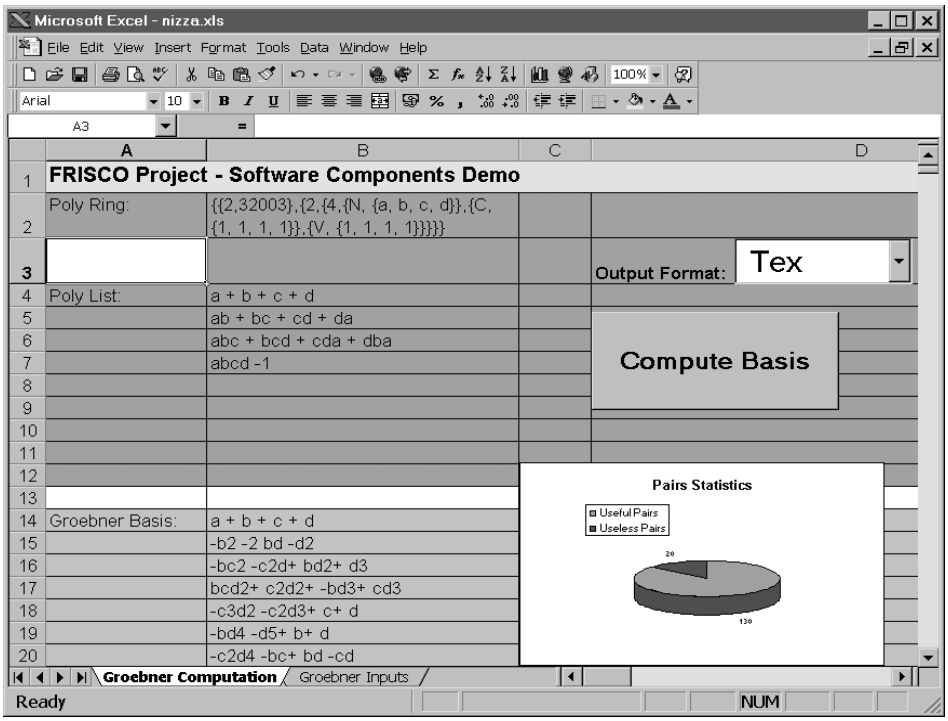

Figure 5: Mathematical component inside Excel

For Unix we use a CORBA implementation based on dynamic modules that can be loaded at runtime. The IDL interfaces are automatically translated into C++ objects. This way it is possible to take advantage of some features of CORBA, such as runtime type information, reference counting, and safe type cast.

Currently we are not using object distribution capabilities of CORBA, since our primary goal is to maximise performance. Thus we are connecting clients and server components through dynamic modules. Accessing a factory is just instantiation of a class and the rest are ordinary method calls. This does not give us the full power of components since for instance it limits us to code written in C++ and compiled with the same compiler, due to the idiosyncrasies of different C++ compilers. However, this problem could be addressed using distributed objects, i.e. keeping the client and the server in two separate addressing spaces and using the ORB to connect them. This way, of course, there is a performance overhead. Note that, since changing our implementation to enable object distribution would require minimum changes.

With ATL components instead it is possible to combine components written in languages such as C, Java or Lisp for which there is an official IDL mapping. An OLE component, written in any programming language, which implements the OpenMath Object specifications, can be used by our mathematical components. The ability to use different languages for clients and servers is indeed one of the major attractions of software components.

9. O^3 tools

To support program development with the framework we developed a set of programming tools:

- an IDL compiler to C++ and to ATL
- an IDL documentation tool, which generates LaTeX documentation from comments in the IDL source
- a visual IDL generator tool, for creating interfaces for OM components, without detailed knowledge of the IDL syntax.

The IDL compiler and documentation tool are based on tools from a public domain implementation [FRE] of CORBA.

The IDL generator is a tool we developed to help specifying interfaces to mathematical components without knowing the precise IDL syntax. Some screen shots are shown in figure 6. It is possible to specify the component name, the set of exported packages and, for each package, the list of exported methods. Parameters and return type for each method can be selected from a list of available types. The IDL generator is not essential for writing components, since the IDL specifications can be written with a text editor. The IDL generator is being further developed and a cross-platform version will be built in Java (currently the generator is available for Win95/NT).

The current version of the O^3 framework is available for Sun Solaris, Linux and Windows 95/NT platforms.

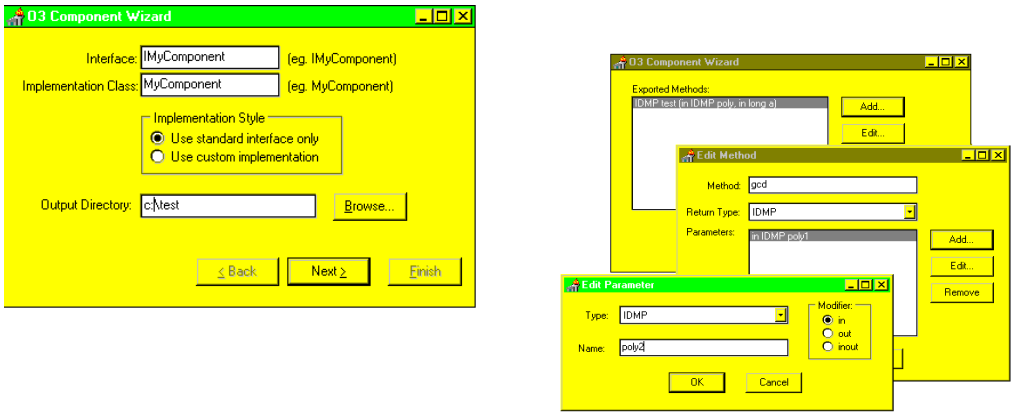

Figure 6: O^3 IDL generator

10. Conclusions and Future Work

We aimed at showing that, once a standard interface for mathematical objects is specified, mathematical software can be developed which is independent from a specific object implementation. This approach goes one step further than current OpenMath implementations, since mathematical objects are not exchanged through a textual representation sent over a data stream. This provides a high level view of mathematical objects and different implementations thereof can coexist within the same program. Furthermore, it is possible to avoid the network overhead by using a component architecture in which different modules can efficiently exchange object references within the same address space.

Our experience may lead to the proposal of an IDL specifications for the OpenMath standard. This will require of course interactions with other groups (such as people working on the PolyMath library) in order to meet different requirements. Interface standardisation is a fundamental requirement to achieve interoperability of components from different sources.

We would like also to experiment the O^3 framework with ActiveX also on Unix, for instance using EntireX DCOM.

In order to prove the viability of the approach, we plan to extend one of the current computer algebra systems with the ability to dynamically load mathematical components, providing a simple mean to use components.

Providing access and support for mathematical components in computer algebra systems is a promising approach to extend their viability.

Acknowledgments

This work was carried out within ESPRIT project FRISCO. We thank prof. Carlo Traverso and Tito Flagella for many useful discussions and suggestions about software components for computer algebra.

References

[ABO] Abbott J., van Leeuwen A., Strotmann A., “Objectives of OpenMath”, Technical Report 12, RIACA, 1996.

[ATL] Grimes, Stockton, Reilly G. V., and Templeman, “Beginning ATL COM Programming”, Wrox Press, 1988.

[BRO] Brockschmidt K., “Inside OLE”, 2nd edition, Microsoft Programming Series, Microsoft Press, 1995.

[COR] CORBA 2.0 Specifications, Object Management Group, http://www.omg.org

[DAL] Dalmas S., Gaetano M., Watt S. M., “An OpenMath 1.0 Implementation”, in *Proceedings of ISSAC’97*, ACM press, July 1997.

[DEN] Denning A., “OLE Controls Inside Out”, 2nd edition, Microsoft Programming Series, Microsoft Press, 1995.

[ENT] Software AG, EntireX DCOM, http://www.softwareag.com/corporat/solutions/entirex

[FRE] Linton M., Pan D. Z., “Interface translation and Implementation Filtering”, *Proceedings of USENIX C++ Conference*, 1994, 227–236

[JOL] The Java OpenMath Library, version 0.3, http://pdg.cecm.sfu.ca/openmath/lib/

[JVB] JavaBeans Specifications, version 1.01, http://www.javasoft.com/beans/docs/spec.html

[MUP] MuPAD – a Computer Algebra System, http://www.mupad.de

[OMA] OpenMath Home Page, http://www.openmath.org

[OPE] OpenDoc, http://www.software.ibm.com/ad/opendoc

[POS] Attardi G., Traverso C., “The PoSSo Library for Polynomial System Solving”, *Proc. of AIHENP95*, World Scientific Publishing Company, Singapore, 1995.

[SOM] SOMobjects Developer’s Toolkit, Programmer’s Guide, available at http://www.software.ibm.com/ad/somobjects

[SOR] Sorgatz A., “Dynamic Modules - The Concept of Software Integration in MuPAD”, in *Proceedings of IMACS'97*, July 1997.