A Lightweight Component Architecture for Efficient Information Fusion*

Olaf Spinczyk, Andreas Gal, Wolfgang Schröder-Preikschat

University of Magdeburg
Universitätsplatz 2
39106 Magdeburg
{olaf, gal, wosch}@ivs.cs.uni-magdeburg.de

January 6, 2001

Abstract

Information fusion is the process of integration and interpretation of data from different sources in order to derive information of a new quality[1]. Software systems for information fusion have to deal with masses of data, which have to be collected, filtered, manipulated, and visualized. Acceptable performance of such systems can only be achieved by a very efficient implementation. At the same time a fusion system is large and complex, thus it should be constructed from components with defined interfaces and responsibilities.

This paper shows that any kind of binary components that need interprocess communication mechanisms and even binary components that run in a common address space have an unacceptable performance when masses of data have to be handled. Instead an architecture of aspect-oriented lightweight components is proposed, that allows the modular construction of information fusion systems from reusable units combined with an overhead-free implementation by omitting context switches and copy operations wherever possible.

1 Introduction

A system for information fusion is large and complex. It consists of several layers starting at the bottom with the appropriate hardware, followed by the operating system, perhaps different database systems, a layer to combine and select data from different sources, fusion algorithms that filter and reduce the datasets, and a user interface to interact with the whole "fusion engine". As an example figure 1 gives an overview of the InFuse[9] system currently developed at the University of Magdeburg.

Several layers of independent and reusable modules are plugged together to build the large and complex final "Workbench for the Information Fusion". This structuring is necessary to get a flexible and maintainable implementation. By using, for instance, CORBA[7]-based components, it is even possible to put the modules on different network nodes, i.e. the fusion system becomes a distributed system.

Using a binary component model such as CORBA for the design and implementation is a very important architectural decision. The positive effects will be as follows:

* This work has been partially supported by the German Research Council (DFG).

[1] This sentence is taken from the Workshop's CfP.

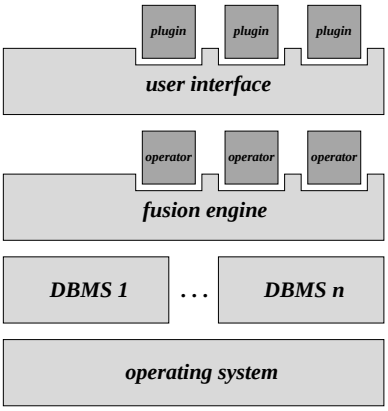

Figure 1: The structure of the InFuse system.

Object Orientation The programming paradigm used for the interaction of large system components fits to object-oriented programming languages that are used very often for the implementation of the components itself.

Distribution transparency Two communicating components use the same interaction mechanism, regardless whether they are located in the same address space, on the same machine, or on different network nodes.

Hardware, operating system, language, and vendor independence CORBA is an open standard that is implemented by many ORB[2] vendors for lots of different platforms and programming languages.

On the other hand there are significant drawbacks that have to be taken into account:

Efficiency CORBA provides lots of features that are not needed in every case. This poses an overhead on the application that might be critical when masses of data have to be handled.

CORBA-ruled source code The source code that is used to implement a CORBA-based component is mainly ruled by predefined CORBA datatypes and calls to CORBA functions. The technical implementation aspect of the inter-component communication is tangled with the algorithms that the component provides. This is, for instance, a problem seen from the maintainability point of view. If the algorithms should be used without CORBA or with some other interaction mechanism, the main algorithms will be hard to isolate. This restricts software reuse.

The following sections of this paper analyze the problems that arise with a binary component architecture (section 2) and propose a concept of aspect-oriented components (section 3) that omits the drawbacks while preserving the positive effects. To illustrate the ideas a concrete example is presented (section 4). The paper closes with a discussion of related work (section 5) and the conclusions (section 6).

2 Analysis

2.1 Efficiency

To measure the overhead caused by binary component communication, an exemplary database application has been constructed (figure 2). The data source component represents a typical relational database. It maintains a large set of records, with each of those

[2] Object Request Broker

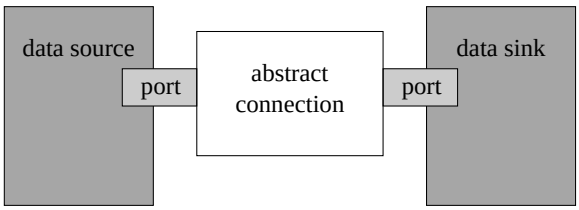

Figure 2: Inter-component communication benchmark environment.

```
interface Field {
  string    name ();  // get field name
  string    value (); // get field value
};

interface Record {
  Field     field (in string field_name); // get selected field
};

interface Recordset {
  boolean   eof();       // true behind the last record
  void      movenext();  // move to next record
  void      movefirst(); // move to first record

  Record    record();    // get current record
};

interface Connection {
  Recordset query(in string statement); // execute query
}
```

Figure 3: CORBA definition of a simple database client interface

records available in the main memory at any time. The client component communicates with the data source to sub-sequentially fetch all records. While this is very simple task, it is a quite common in real-world applications and a useful indicator to measure the data transfer bandwidth between data producer and data consumer. In an information fusion workbench large volumes of data have to be transfered between the components, thus the communication latencies are essential.

In this example we utilize a simple object-oriented interface (figure 3) that follows Microsoft's Active Data Object technology and SUN's Java Data Access API.

The client/server communication is benchmarked using several implementations of the CORBA middleware and additionally with a lightweight hand-coded communication library for UNIX sockets and TCP/IP. For the two later methods a simple, synchronous protocol has been defined. It adds nearly no marshaling overhead and is thus well suited to measure the raw communication performance over address space boundaries, which is the lower limit that a middleware could ever reach. As CORBA does also support communication of components in the same address space, the overhead for this is measured, too. It is compared with the communication performance of a direct data access on the source code level without and with a single copy operation.

In this benchmark the class of CORBA ORBs is represented by the freeware MICO implementation and the commercial omniORB. Other middleware implementations like

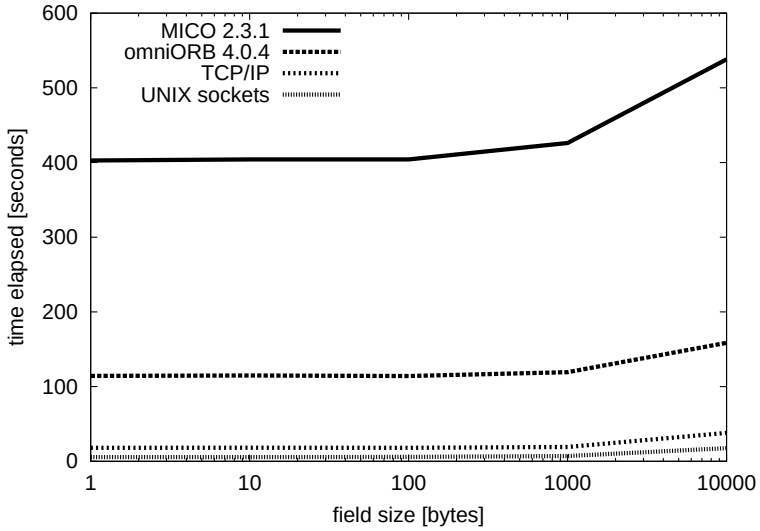

Figure 4: Component communication over address space boundaries.

Microsoft's DCOM have not been examined, because comparing different middleware implementation is out of the scope of this paper and we do not expect significant differences. C++ is used as implementation language because it is widely used in combination with CORBA. Using other languages is expected to produce similar results.

All measurements have been done on a Pentium III PC[3]. The machine was not under load during the tests and GNU g++ [4] was used to compile all examples. In case of TCP/IP client/server communication both instances ran on the same machine using a virtual loopback network device.

To explore the influence of data copy operations on the results, the benchmark has been conducted using different field sizes from 1 byte to 10 kilobytes. 100000 fields were fetched sequentially from the server.

It should be remarked that it is neither the interface nor the components we would like to benchmark. The focus lies on the communication infrastructure that is used to implement the interface.

Figure 4 compares different implementations of component communication over address space boundaries. While omniORB performs significantly better in this test than MICO, even omniORB is still approx. 6 times slower than raw TCP/IP communication. Using UNIX sockets for communication would further reduce the overhead to about one third. Thus, using CORBA for communication between objects located in separate address spaces is much slower than the performance of the raw communication interface would suggest.

In figure 5 the fastest communication method over address space boundaries (UNIX sockets) is compared with local communication. Both CORBA implementation show still a serious communication overhead that roughly corresponds to the communication performance of UNIX sockets. The direct data access candidate is approx. 100 times faster than the fastest CORBA representative (MICO), while for large field sizes the version with an additional copy operation is still twice as fast as CORBA in the same address space and 20 times faster than CORBA over address space boundaries. The CORBA overhead for small field sizes (up to 1000 bytes) is dominated by the general costs of a CORBA remote object invocation. Only for very large field sizes the cost to transport the data itself becomes a

[3] 600 Mhz, 192 MB of RAM and Linux with kernel version 2.2

[4] Version 2.95.2 has been used with -O6 optimization settings. Vendor delivered libraries have been compiled with vendor defined default settings.

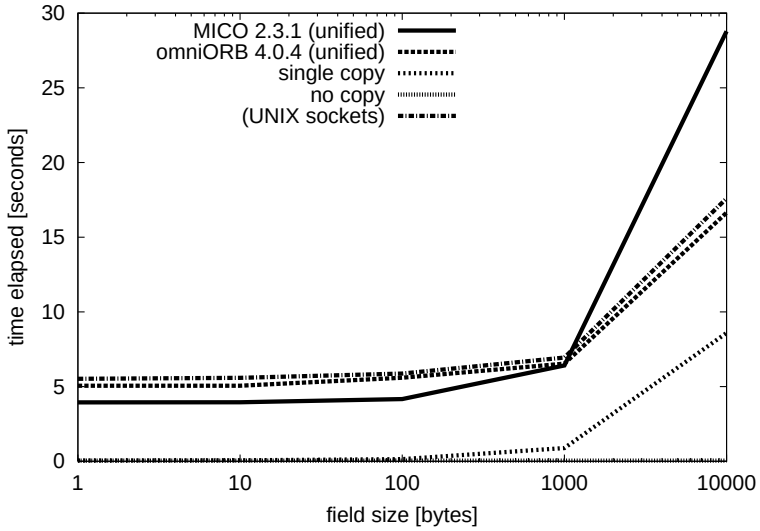

Figure 5: Component communication in the same address space.

significant cost factor. The measurements show also, that CORBA has a serious overhead even when used for object communication in the same address space.

2.2 Code Tangling

Using a binary component model like CORBA does not only have an impact on the run-time behavior of an application. The CORBA middleware, for instance, also requires the programmer to follow certain implementation guidelines. As the implementation language C++ does not directly support distributed computing, CORBA adds a number of new data types to distinguish between input and output variables and local or remote object references. CORBA does also dictate how strings are passed as return values as they have to be dynamically allocated with CORBA specific allocation handlers. To establish object communication, the programmer has to use CORBA specific library calls. As a result the code of an application using CORBA is dominated by CORBA specific expressions (figure 6). CORBA was designed with portability in mind. Code written according to the CORBA specification should be usable with every CORBA ORB following the standard. This has been only partially reached, because most ORB vendors extend the standard with incompatible features or only partially implement it. Simply switching from one ORB to another is not possible today. With this level of code tangling it becomes extremely expensive to switch over to a different middleware like DCOM. Code maintenance is also difficult and reusing the code in non-distributed environments means a lot of error prone code editing by hand.

2.3 Results

The discussed run-time and design-time overhead makes middleware concepts like CORBA inappropriate for component communications in systems with many components and high communication demands like information fusion systems. The loss in performance when components are located in separate address spaces is so severe, that moving components with intensive communication relations into the same address space seems mandatory. But even in this case, the interface has to be designed CORBA-aware, i.e. by reducing the total amount of remote object invocations and using larger chunks of data to achieve acceptable performance. From the design perspective this is unwanted as solely

```
...

static void hello(Echo_ptr e)
{
  if ( CORBA::is_nil(e) ) return;
  CORBA::String_var src = (const char*) "Hello!";
  CORBA::String_var dest = e->echoString(src);
  cerr << (char*) dest << endl;
}

int main(int argc, char** argv)
{
  CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "omniORB3");
  CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
  PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);
  PortableServer::POAManager_var pman = poa->the_POAManager();
  pman->activate();
  Echo_i* myecho = new Echo_i();
  Echo_var myechoref = myecho->_this();
  myecho->_remove_ref();
  hello(myechoref);
  orb->destroy();
}
```

Figure 6: Extract from a hello world example for omniORB

the problem and not the middleware should drive the design process. But also for run-time performance reasons transporting large data chunks is not effective, as it leads to a client side data cache, which does not only consume memory but also complicates the use of the interface.

3 Lightweight Components

The efficiency problem can be solved by using CORBA interfaces only where it is absolutely necessary, i.e. where address space boundaries have to be passed. If the communicating parts of the system are in the same address space they can be compiled and linked together and the efficiency benefits from the whole set of compiler optimizations and omitted copying and context switches. This means that the components have to be provided in source code (at least partially).

CORBA-based source code components will not solve the problems discussed in section 2, because the CORBA semantics implicates at least one copy operation. The measurements show that today's CORBA implementations come with even more overhead. This means that data duplication will not disappear and the code tangling problem remains, too.

Our preferred solution to implement the required behavior follows the idea of aspect orientation. The next section 3.1 will discuss this aspect-oriented approach. The resulting aspect-oriented distributed object model is presented in section 3.2. Section 3.3 describes how source code components can be developed and deployed independently.

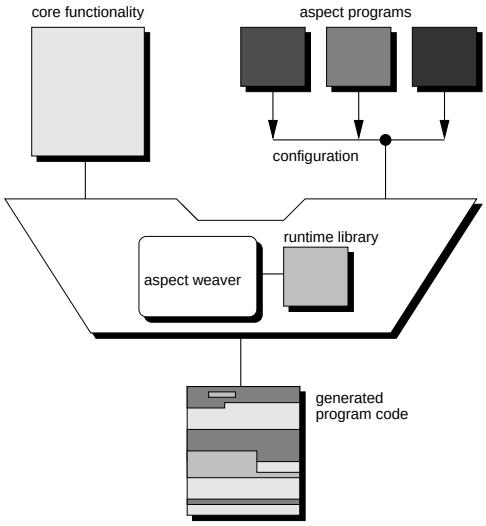

Figure 7: Large scale configuration by aspect switching

3.1 Aspect-Oriented Approach

Aspect-oriented programming (AOP) [5] means separation of concerns in the implementation where standard programming language composition mechanisms do not allow this. The distribution aspect of an application is a good example. With an implementation language like C++ and CORBA middleware, for instance, most parts of a distributed application are normally distribution-aware. This is not only a question of CORBA defined data types and functions but of a different meaning of pointers, procedure calls, and asynchronous behavior. The language does not provide adequate mechanisms to encapsulate the implementation of this aspect. This leads to the code tangling problem. The AOP approach is to provide a problem adequate “aspect language” that is used to implement the aspect separately. Because the compiler that generates the final code needs both parts merged, there is an aspect specific tool - the “aspect weaver” - that takes both parts and “weaves” them together. Conceptually this can be done at compile-time or at run-time. The second would be necessary if the distribution scheme of an application changes at run-time. This is not needed in our case, but by selecting different aspect programs at compile-time, it becomes quite easy to configure the system according to requirements on static distribution schemes. Figure 7 illustrates this concept. The run-time library will typically be needed for the implementation of the high-level features of the aspect language.

The main idea of lightweight components for information fusion systems is that they are provided as source code, which is implemented without having multiple address spaces, i.e. distribution, and all the consequences like synchronization in mind. This allows an overhead-free connection of two components with the standard programming language mechanisms. If a configuration is required where the same two components have to run in a different address space, the necessary code modifications are done with the help of an aspect weaver. A lot of component specific information is needed to drive this transformation step. This is provided by the component developer in separate description files. With the tool support presented in this section the code tangling and the efficiency problem can be solved without reducing the number of possible configurations compared to “classic” component models.

3.2 Distributed Object Model

If a component is configured to work distributedly, the distribution scheme has to be specified. The object-oriented paradigm is not inherently limited to communication between local objects. The conceptual passing of messages between objects directly reflects communication in distributed environments. Thus an object model is well suited to describe the distribution scheme by associating objects with address spaces.

One reason to implement a component in a distributed way is that there are cases where parts of a component depend on resources that are only available in a certain address space. For example, parts of a database component rely on files, which are located on a specific network node, while the client side of the component can be executed in any address space. Thus we have to provide means to bind the execution of objects of a specific class to a certain address space. This leads to client/server relationships.

Another reason is to delegate workload to parallel executing machines. For example, if some specific entry in a binary tree is searched, the tree could become a distributed data structure and the operations on subtrees are executed at the same time. This means that the objects of the same class must be executed on different nodes. The following list describes the different object execution modes that we have identified.

bound An object with "bound" excution mode (or bound object for simplicity) is always executed in a specific address space. This kind of mode is needed where the execution requires some specific resource that is only available in the address space where the execution is bound to.

rooted The execution of a rooted object always happens in the address space where the object was instantiated. This can be used to construct distributed data structures.

mobile The mobile execution mode means that the address space where the object is executed can change at run-time.

local Local execution means, that a method invocation always happens in the address space of the caller.

In our prototypical implementation of the presented concept we associate these execution modes with the objects of the classes that the component consists of. This happens in a separate distribution describtion (the distribution model). Depending on performance or security requirements different distribution models might be useful. For example, the recordset resulting from a database query can be transfered to the client at once or record by record. To select the second behavior the "Recordset" class must be bound to the server side. Otherwise the "local" execution mode will be used, because it is default in our implementation. The configuration of the distribution aspect is done by selecting one of these distribution models, which can either be delivered by the component developer or implemented by the component integrator.

The distribution model does not depend on the middleware platform, which is used for the actual remote method invocations. All address spaces are specified by abstract identifiers. The definition, which kind of server has to manage this address space, is part of the integrator's requirement defintion.

Many more information has to be provided to generate a distributed program from a local one. A lot of this information is only required to improve the performance but is irrelevant if the goal is only to generate a running system. This information is specified by so called "hints". For example, it can be specified that only parts of objects need to be transfered or if they have to be transfered at all.

Other required information like the size of arrays in our supported implementation language C++, which identifies arrays by pointers to the first element, are obtained by a combination of weaving techniques and a run-time support library.

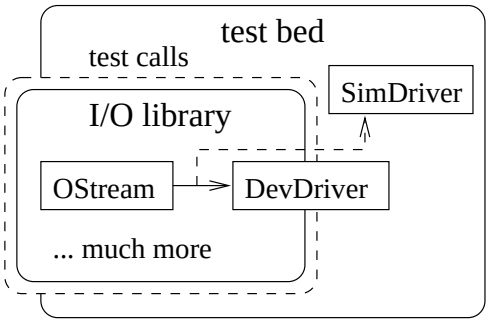

Figure 8: A component test bed

Section 4 will provide an example that gives an overview about the mentioned aspect programs that describe distribution models and the code transformations that are necessary to weave the distribution aspect into a single address space program.

3.3 Source Code Components

Many relevant scientists in the field of component software argue that components have to be binary[10]. Others do not accept this restriction and it is also questionable what "binary" means (e.g. is Java bytecode binary?). Widely accepted is the requirement that "a software component can be deployed independently and is subject to composition by third parties" [11]. In our opinion with source code components this is possible quite well, if an adequate infrastructure is provided.

Problems with independent development and testing of a component arise, when the component needs to delegate responsibilities to some other component. As an example consider an I/O library that provides abstractions for input and output streams, buffering strategies, and so on. For the actual output on a physical device the component is not responsible. Instead, it requires some other component that implements a device driver interface. The implemented source code component uses this interface, thus function names, parameter types, and so on can be derived from the source code. In an object-oriented environment the required interface can elegantly be represented by a "dummy" class, that only declares methods but does not implement them. For the development a test bed is needed which provides a class that is compatible with the "dummy" class. A tool is used to transform the component in a way that all references to the dummy class are replaced. This tool must also check for compatibility. Figure 8 shows the I/O library component plugged into a development and test bed.

The same replacement mechanism is used when an application is composed from many independently developed components. Automatic renaming is applied to avoid name clashes. With these simple manipulation operations systems can be composed without the need to edit the source code by hand. Thus, a "robust" integration can be achieved.

4 Example

Figure 9 presents a small example that shows the input files of the aspect weaver and the results after it was running. The weaver is used to generate the necessary code for the distribution aspect of a component. Client and server are implemented with normal C++ code. They share the result of a database query (a "Recordset") by using references to the same memory region: no copying is needed. The aspect code consists of three parts: the component description, which describes the classes that are part of the component, the distribution model, and the requirement defintion, which is typically implemented by the

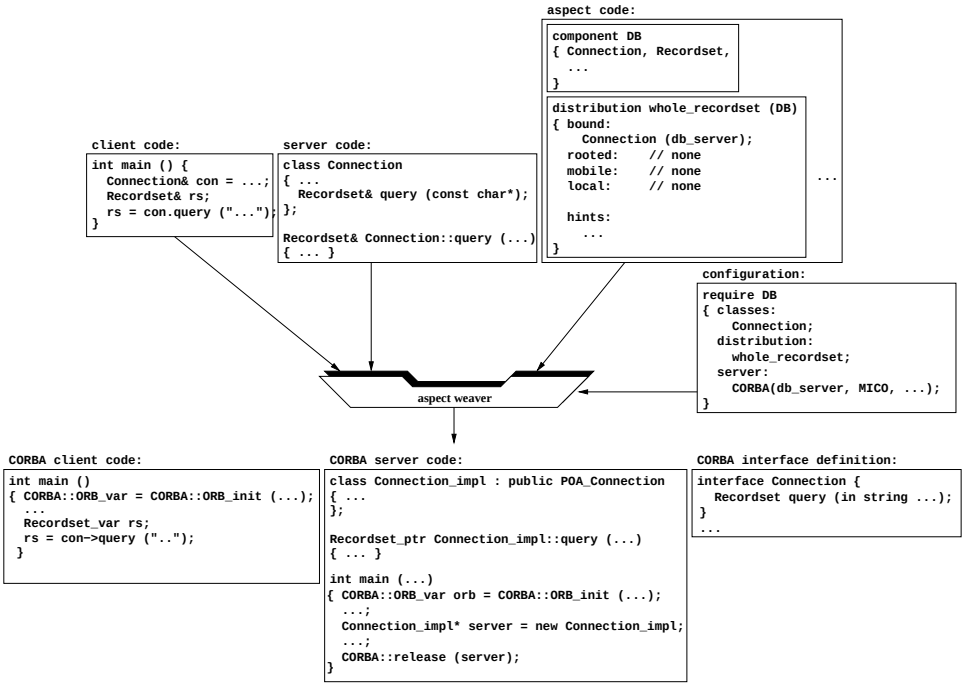

Figure 9: Using an aspect weaver for the generation of component interfaces

integrator to configure a component.

In the example the selected distribution model requires that the class “Connection” is bound to an abstract address space “db_server”, because the database files reside on a specific node. All other objects are executed locally by default. The result is that the client-side reference “con” will become a remote reference and the call “con.query” will become a remote method invocation. In the requirement definition a CORBA-based server is associated with the abstract address space “db_server”. This means that the remote accessible classes have to be parsed, analysed, and an IDL file has to be generated that is needed to build the CORBA stubs and skeletons. The original client and server code have to transformed in a way that they now interact via a CORBA middleware.

5 Related Work

The work presented in this paper was mainly inspired by the concept of aspect-oriented programming. It was first presented at ECOOP’97 and is a topic of interest in the software engineering and OO community until today. The aspect-oriented implementation of distributed systems is the purpose of the aspect language D[6]. It is restricted to Java components and can be seen as a starting point for further research, because it covers only a minimal subset of the distribution aspect. For example, quality of service in distributed systems can be modeled as an aspect[2].

Some other groups are also working on the topic of distributed software construction with AOP concepts. A-TOS[8], for example, is a reflective framework that introduces the concept of “aspect components” for this purpose. The system uses event interception and wrappers to implement the distribution aspect in TCL-based applications. As far as we understand A-TOS is not well suited for information fusion systems, because of our requirement on high efficiency. Lots of problems in our project are related to the component implementation language C++ and its pointer concept which does not exist in TCL. It is our goal to solve these problems, because pointers give programmers the chance to let different

parts of a program share data.

The concept of grey-box connectors [1] was proposed to model abstract connections of communicating source code components. Here the implementation of concrete connections becomes an aspect. Our work follows this approach, but goes much more into technical details, for example by defining the distributed object model.

6 Conclusions and Future Work

Application run-times have to be ruled by the complexity of algorithms and hardware limitations instead of artificial boundaries forced by a specific system architecture. This can be achieved by the presented concept.

Closed components like database systems or the operating system are always part of the fusion system, thus the application of lightweight components is restricted to self-made code. An even larger impact on the performance is possible if the used database systems would be reusable source code components as well. At least for open source database systems this is possible and would make them more competitive compared to their commercial counterparts. The goal is to create a fusion system where “it is the system design which is hierarchical, not its implementation”[4].

The work presented is this paper is work in progress. An aspect weaver environment is working that is responsible for the actual code manipulations in C++ source code components and the execution of aspect weaver plugins. Currently our most sophisticated application of this environment is a weaver that allows the restructuring of class relations in components[3]. The automatic generation of CORBA IDL files and component implementations from C++ classes is currently in a prototypical state. Especially handling of communication errors is completely missing. It is planned to allow error handling in an aspect-oriented manner, i.e. error handling code will be separated from the component code.

Future work will be to finish the implementation. Our hope is to find a better way to implement distributed software systems with an object-oriented programming language like C++. Aspect orientation is the key concept for this task. The next logical step after making sequential local code to non-sequential distributed code is to make the same code to parallel code that can be executed efficiently on parallel hardware platforms. Here again information fusion will be an interesting application.

References

[1] U. Aßmann, T. Genßler, and H. Bär. Meta-programing with Grey-box Connectors. In *Proceedings of the 33rd International Conference on Technology of Object-Oriented Language (TOOLS 33)*. IEEE Computer Society, June 2000. ISBN 0-7695-0731-X.

[2] C. Becker and K. Geihs. Quality of Service — Aspects of Distributed Programs. In *Proceedings of the ICSE’98 Workshop on Aspect-Oriented Programm ing*, 1998.

[3] M. Friedrich, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. Efficient Object-Oriented Software with Design Patterns. In *Krzysztof Czarnecki, Ulrich W. Eisenecker (Eds.): Generative and Component-Based Software-Engineering. First International Symposium, GCSE’99*, Erfurt, Germany, Sept. 1999. Springer-Verlag. Revised Papers. LNCS 1799.

[4] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. Technical Report SPL97-008 P9710042, Xerox PARC, Feb. 1997.

[6] C. V. Lopes and G. Kiczales. D: A Language Framework for Distributed Computing. Technical Report SPL97-010 P9710047, Xerox PARC, Feb. 1997.

[7] Object Management Group. `http://www.omg.com/`.

[8] R. Pawlak, L. Duchien, G. Florin, L. Martelli, and L. Seinturier. Distributed Separation of Concerns with Aspect Components. In *Proceedings of the 33rd International Conference on Technology of Object-Oriented Language (TOOLS 33)*. IEEE Computer Society, June 2000. ISBN 0-7695-0731-X.

[9] K.-U. Sattler and G. Saake. Supporting Information Fusion with Federated Database Technologies. In *Proceedings of the 2nd Int. Workshop on Engineering Federated Information Systems (EFIS '99)*, Kühlungsborn, Germany, 1999.

[10] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison Wesley, 1997.

[11] C. Szyperski and C. Pfister. Workshop on component-oriented programming. In *ECOOP 1996 Workshop Reader*, 1997.