ATTRIBUTES &CO – COLLABORATIVE APPLICATIONS WITH DECLARATIVE SHARED OBJECTS

Eva Kühn and Fabian Schmied
Institute of Computer Languages, Vienna University of Technology
Argentinierstraße 8, A-1040 Wien

ABSTRACT

When compared to local applications, aspects such as data transfer, transaction handling, and change notification add an additional level of complexity when developing collaborative distributed applications. Middleware layers strive to reduce this complexity, but without the right application programming interface, most of the complexity is only relocated instead of being removed. ATTRIBUTES &CO, a language binding for space-based middleware layers, accomplishes to reduce the complexity of distributed software development by employing the declarative mechanisms provided by modern object-oriented languages. Using the paradigm of declarative programming for describing the properties of distribution, ATTRIBUTES &CO helps to improve the development process, raising code quality and reducing development time.

KEYWORDS

Distributed applications, collaboration, languages, .NET, attributes, metadata

1. INTRODUCTION

Collaborative, distributed systems, when compared to local, single-user applications, introduce additional complexity [Lopes and Kiczales, 1997]. The clients participating in the system have to be coordinated, data has to be held in a consistent state, and network deficiencies have to be coped with. To overcome this raised level of complexity, *middleware layers* provide services that hide the details of distribution, such as network communication, transaction handling, or data persistency. As an example, *virtual shared memory (VSM)* middleware systems [Kühn, 2001] (e.g. JavaSpaces [Sun, 2003]) facilitate application development via stateful communication, hiding the details behind *space-based computing:* distributed applications can put data objects into a virtual shared object space and collaboratively work on them. Sophisticated space-based middleware layers also have support for services such as nested transactions, automatic change notifications, and intelligent peer-to-peer data distribution (e.g. CORSO [Kühn and Nozicka, 1998], XVSM [Kühn, 2005]).

Mechanisms for data replication and consistency, vital to generalized distributed applications as well as collaborative systems, have been widely researched in the context of middleware and groupware [e.g. Lukosch, 2002], but the interfaces provided to access those mechanisms are often neglected.

Because modern distributed applications are usually developed in object-oriented programming languages (e.g. Java or C++), middleware layers have to integrate with these. Language-independent middleware platforms such as CORBA or CORSO usually provide thin application programming interfaces (APIs), or *language bindings*, which can be used to interface the middleware from object-oriented programming languages, offering low-level access to the middleware's services.

Of course, object-oriented programming (OOP) is a proven way for developing large-scale applications; powerful platforms and frameworks, best-practices, and design patterns facilitate the creation of high-quality software applications and components. However, one should be aware that OOP is not a panacea. Basically, object-oriented programming is an imperative programming paradigm: a program tells the computer exactly, step by step, what it should do. While many problems are well-suited for imperative solutions, some are not.

In particular, we argue that employing the imperative programming paradigm for middleware language bindings leads to large and unnecessarily complicated programs, forcing the programmer to concentrate on

side issues, thus introducing an additional level of complexity to the application developer – quite contrary to the original goal of reducing complexity [Schmied and Kühn, 2004].

As a solution, we introduce ATTRIBUTES &CO (“attributes and coordination”), a declarative language binding for VSM-based middleware, based on the facilities provided by modern object-oriented programming languages. We demonstrate how the declarative concept enables programmers to add the aspects of distribution to their programs in a clean and easily understandable way, allowing them to concentrate on the problems at hand. While retaining independence from programming language and programming platform, this reduces the amount of code needed for distributed applications, allows for better separation of concerns, provides better code quality, and thus improves the overall development process. ATTRIBUTES &CO can be used for any kind of distributed application, but in this paper, we demonstrate its use for collaborative applications.

2. INTRODUCING COLLABORATION TO AN APPLICATION

We illustrate the use and the workings of ATTRIBUTES &CO via an example: *Whiteboard* is a peer-to-peer C# application for the collaborative creation of drawings. Its functionality is simple: the application displays a drawing canvas, onto which one can draw shapes, such as lines, rectangles, or text. Via a network connection, multiple users (*peers*) participate in the drawing process, erase the canvas or parts of it, and immediately see what others have drawn.

As a beginning, we define a class hierarchy for the drawing itself and the shapes it contains, as shown in Figure 1. Since the drawing and the shapes comprise all the data shared among the peers, all communication functionality should be implemented within the context of these classes. The actual Whiteboard is implemented as a graphical user interface application, which uses the classes shown here as its data model. The application connects to the network and creates the concrete *IDrawing* object, but apart from that, it is totally agnostic of distribution or middleware.

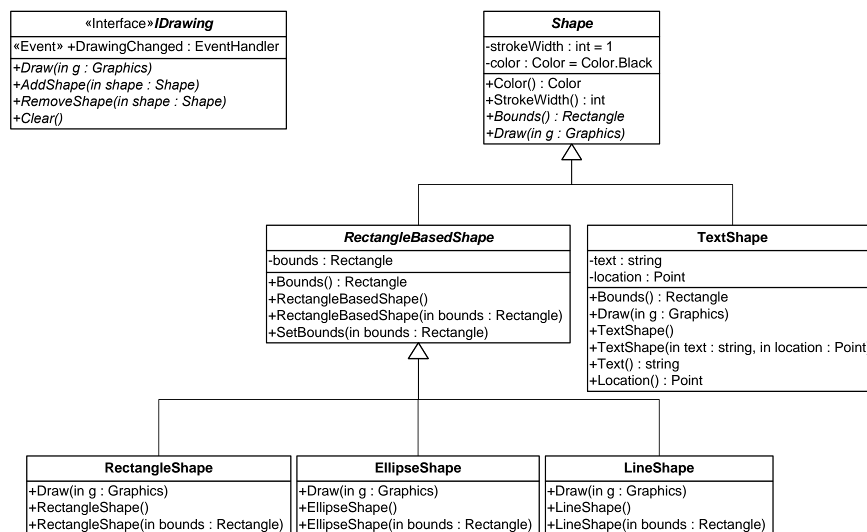

Figure 1. Whiteboard class hierarchy

We begin with a local, single-user implementation, which is quite straight-forward: there are a number of different *Shape* classes, all of which support certain general (*Bounds, Color, StrokeWidth*) and special (*Text, Location*) properties, as well as a *Draw* method for drawing them on the screen. The implementation of *IDrawing* simply serves as a container for them (allowing addition and removal), delegating all drawing requests to the respective shapes. When the drawing is changed, the *DrawingChanged* event is fired, on which the application can react and repaint the shapes. Listing 1 shows excerpts from the *LocalDrawing* class, which is an implementation of *IDrawing*.

For the transition from the local, single-user Whiteboard implementation to a distributed, collaborative, multi-user application, the following additional demands must be fulfilled:

- Shape data needs to be transferred over the network between the clients of the application (*data transmission*).
- Whenever a user changes the drawing, all peers should be informed in near-time and the *DrawingChanged* event must be fired (*change notification*).
- Simultaneous modifications of the drawing have to be synchronized, so that the data is always consistent (*transaction safety*).

For the last point, consider two peers simultaneously doubling the size of a shape: this requires the operations of reading the most recent shape data from the network, changing the object's size, and sending the drawing back to the network. If these operations are not synchronized (e.g. by obtaining an exclusive lock on the object using a distributed transaction mechanism), inconsistencies will occur and the resulting size of the shape cannot be predicted. This is a classic problem of databases, a solution to which is adherence to the *ACID* principle (atomicity, consistency, isolation, durability) [Gray and Reuter, 1993].

Implementing these requirements using classic message-based technologies (e.g. sockets or message queues) would be very complicated; especially transactional safety is a non-trivial task to solve. Fortunately, advanced middleware layers already provide the services that are needed to fulfill these demands.

```
public class LocalDrawing : IDrawing {
  public event EventHandler DrawingChanged;
  private ArrayList shapes = new ArrayList();

  public void Draw(Graphics g) {
    foreach (Shape shape in shapes) {
      shape.Draw(g);
    }
  }

  public void AddShape(Shape shape) {
    shapes.Add(shape);
    OnDrawingChanged();
  }

  private void OnDrawingChanged() {
    if (DrawingChanged != null) {
      DrawingChanged(this, null);
    }
  }

  // RemoveShape, Clear analogous to AddShape
}
```

Listing 1. Local Whiteboard implementation example code

3. IMPERATIVE IMPLEMENTATION

Listing 2 shows excerpts from an implementation of a collaborative version of the Whiteboard application. It uses the object-oriented .NET language binding provided by CORSO, a VSM middleware implementing the space-based paradigm. Although space-based computing offers a high-level abstraction and leads to applications with fewer lines of code than using message passing, the resulting code of our example is very long and much harder to understand than the original code of Listing 1: it introduces additional methods for dealing with the concerns of data transmission (*Refresh, Persist, Read, Write*) and change notification (*NotificationThread*), and it also changes existing methods (e.g. *AddShape*) to integrate transaction handling.

Since the scope of this paper is not CORSO's language binding, we will not discuss the whole listing in detail. However, for a representative example, consider the method *AddShape*. In the original implementation, this method had to deal with one concern: the addition of shapes[1].

[1] Actually, *AddShape* also had to deal with the concern of change notification. However, this was cleanly encapsulated and could not be fully decoupled from the main concern.

Now, an additional, orthogonal and unrelated concern has to be handled: transaction safety. Fortunately, the middleware already offers mechanisms to create transactions, to read and write data objects in their context, and to commit them. However, *AddShape* clearly shows that the imperative paradigm is not a natural way to implement collaboration. Complicated code constructs are needed to achieve the required results and the code is hard to understand because it tangles several concerns beyond the original problem to be solved.

```
public class SharedDrawing : IDrawing,
    CorsoShareable {
  public event EventHandler DrawingChanged;
  private ArrayList shapes = new ArrayList();
  private CorsoConnection con;
  private CorsoVarOid oid;

  public SharedDrawing(CorsoVarOid oid) {
    this.oid = oid;
    con = oid.GetConnection();
    // prepare near-time notification thread
    Thread t = new Thread(new
      ThreadStart(NotificationThread));
    t.IsBackground = true;
    t.Start();
  }

  // refreshes object in a transaction
  private void Refresh(CorsoTransaction tx) {
    oid.ReadShareable(this, tx);
  }

  // persists object in a transaction
  private void Persist(CorsoTransaction tx) {
    oid.WriteShareable(this, tx);
  }

  // deserializes object data
  public void Read(CorsoData data) {
    int elements = data.GetStructTag(…);
    shapes.Clear();
    for (int i = 0; i < elements; ++i) {
      shapes.Add(Shape.DeserializeFrom(data));
    }
  }

  // serializes object data
  public void Write(CorsoData data) {
    // ...
  }

  // adds a shape, ensuring transaction safety
  public void AddShape(Shape shape) {
    CorsoTopTransaction tx =
      con.CreateTopTransaction();
    try {
      Refresh(tx);
      shapes.Add(shape);
      Persist(tx);
      tx.Commit(…);
    }
    catch (CorsoException e) {
      // handle error accordingly
    }
  }

  // provides near-time change notification
  private void NotificationThread() {
    CorsoNotificationItem item =
      new CorsoNotificationItem(…, oid);
    ArrayList items = new ArrayList();
    items.Add(item);
    CorsoNotification notification =
      con.CreateNotification(items, …);
    while (true) {
      try {
        CorsoData data = con.CreateData();
        CorsoNotificationItem fired =
          notification.Start(1, data);
        if (fired == item) {
          Read(data);
          OnDrawingChanged();
        }
      }
      catch (CorsoException e) {
        // handle accordingly
      }
    }
  }

  // RemoveShape, Clear analogous to AddShape
}
```

Listing 2. Shared drawing with imperative language binding

4. DECLARATIVE WHITEBOARD

Object-oriented programming is well-suited for implementing the core functionality of the Whiteboard application. However, as seen in the previous section, it is obviously not a good choice for integrating the orthogonal concerns of distribution at the source code level. The ideal way of expressing these would be a declarative one: *declare* what data should be transferred to other peers (and how), what events should trigger notification actions, what the transactional semantics of certain operations should be. Generally, in *declarative programming* it is not specified what the computer should do to reach a solution. Instead, enough information is declared for the computer to infer the solution on its own [Wikipedia, 2005].

ATTRIBUTES &CO allows for the combination of declarative programming with that of object-oriented programming for distributed application development by using a feature provided by modern object-oriented programming languages and platforms: *extensible metadata* [Austin, 2004] [ECMA, 2002]. Using extensible metadata, as provided by .NET and Java 1.5, source-code entities, such as classes, methods, or fields, are annotated with special *attribute objects*, which can be analyzed at compile-time or load-time (using tools like MEXLIB [Schmied, 2003]) and at runtime (using *Reflection* [Forman and Forman, 2004]).

Note that ATTRIBUTES &CO is *not* a middleware system: it does not solve any of the low-level problems of distribution or collaboration. Instead, it provides a way for *interfacing* with middleware layers. ATTRIBUTES &CO provides a number of attribute classes, which are used to express the concerns of distribution in a declarative fashion. Either at compile-time or at runtime, ATTRIBUTES &CO analyzes the objects used in the application and ensures that middleware services are invoked at the right time.

Listing 3 shows an implementation of the Whiteboard application, which adds the facilities of collaboration via ATTRIBUTES &CO. The meaning of the different source code annotations (bold-face code) is explained in the rest of this section.

The power of ATTRIBUTES &CO, in combination with the VSM-based middleware, is its concept of transparent *shared business objects*. A shared object looks and feels just like any ordinary, local object instance in the object-oriented programming language of choice, the only difference being that multiple peers on the network can have a reference to it at the same time. Peers can access the object simultaneously in a transactionally safe way, and each peer is immediately informed about any changes. Behind the scenes, ATTRIBUTES &CO associates objects of the programming language with data located in the virtual shared memory, automatically applies (de)serialization and transaction mechanisms, and handles notifications as necessary (see the following subsections).

Shared objects are totally independent of language or platform: the same shared object can be used from Java applications as well as .NET applications, on Windows platforms as well as on systems running Linux, and they can even be used from mobile devices. Components that use shared objects can also interoperate with applications implemented via traditional language bindings (e.g. with C/C++), because ATTRIBUTES &CO allows customized data serialization for such legacy data objects.

Listing 3 shows how the concepts of shared objects are expressed declaratively, at the class source code level. To specify that the *SharedDrawing* class should be used for shared objects, it is simply marked with the *SharedObject* attribute. In principle, this annotation is all preparation needed to provide objects of that class with an identity in the distributed application. In particular, it is not necessary to inherit from a certain base class or implement any interfaces, leaving the application architecture as flexible as possible.

Instantiation of shared objects is handled by a special factory object provided by ATTRIBUTES &CO, which encapsulates the object retrieval facilities – e.g. peer-to-peer object querying – provided by the underlying middleware layer. (This is not shown in the listing.)

```
[SharedObject]
[AutoRefreshed]
[UIErrorPolicy]
public class SharedDrawing : IDrawing {
  public event EventHandler DrawingChanged;
  [PolymorphicListSerializer]
  private ArrayList shapes = new ArrayList();

  [Transactional]
  public void AddShape(Shape shape) {
    return shapes;
  }

  [OnAutoRefresh]
  private void OnDrawingChanged() {
    if (DrawingChanged != null) {
      DrawingChanged(this, null);
    }
  }

  // RemoveShape, Clear analogous to AddShape
}
```

Listing 3. Distributed Whiteboard with ATTRIBUTES &CO

4.1 Shared Object State and Behavior

Typically, a shared object will hold state to be distributed within the application. State is added to a shared object by adding fields to its class. When the object data is sent over the network, the fields are serialized to binary form and transferred to the peers sharing the object.

ATTRIBUTES &CO provides automatic data serialization and has a number of predefined serializer objects. The serialization process can be influenced via a variety of attributes, e.g. fields can be marked not to be serialized, custom serializers can be chosen for object members, and serialization order can be defined to allow for integration with legacy applications. Altogether, this comprises a very powerful serialization mechanism, which is still easy to understand and can be integrated across languages and platforms.

As opposed to e.g. JavaSpaces, ATTRIBUTES &CO does not use the automatic serialization mechanisms provided by the programming platform or language. The reason for this is to remain platform-independent as well as to overcome any restrictions imposed by these mechanisms (such as field ordering, restriction to serialization of public fields, etc.).

In Listing 3, the drawing's member *shapes* is marked to use a special serialization object (*PolymorphicListSerializer*) which supports polymorphic usage of shapes. Similarly, serialization of shape objects (not shown in the listing) has to be configured declaratively.

In an object-oriented application, shared objects will not only need to have state, but also behavior to access and manipulate it. ATTRIBUTES &CO allows specification of behavior simply by adding methods to a shared object's class. There are two kinds of business methods supported by ATTRIBUTES &CO:

- *Read-only* methods are used to read a shared object's state without modifying it.
- *Update* methods are used to modify a shared object's state.

Update methods usually change the state of shared objects based on the former state of the same or other objects. Since multiple peers can access the objects at the same time, the situation of two or more peers manipulating them simultaneously is quite possible and even likely. Without proper mechanisms, this can lead to inconsistency problems.

The ACID principle is supported by ATTRIBUTES &CO via *transaction contexts*. The first call to an update method causes a transaction context to be created. All changes made to shared objects within that context (also via other update methods, i.e. nested transactions are possible as well) are performed on the most current state of the objects involved and guaranteed to be executed in isolation. The changes are checked for consistency when the initial update method exits. If consistency can be guaranteed, the context is committed and the changes as a whole are made durable. If not, they are undone by ATTRIBUTES &CO. The responsibility for deadlock avoidance, for example by using optimistic locking strategies [Kühn, 1994], is left to the middleware layer used by ATTRIBUTES &CO.

In Listing 3, the method *AddShape* is marked to be an update method by addition of the *Transactional* attribute. All code executed within the method body of *AddShape* is guaranteed to operate on the most current state and to be performed either as a whole or not at all.

4.2 Handling Errors

In comparison with a local application, there are a number of additional error conditions which can occur in a distributed application. Network timeouts, for example, can prevent object data from being transmitted correctly, concurrency conflicts can stop a transaction from being successfully committed, and so on. Often, the user of a shared object does not expect such exceptions to happen, and, although it would be possible to wrap every method call to a shared object in a *try/catch* block, the handling of these errors should not be the responsibility of the caller. Therefore, ATTRIBUTES &CO provides the notion of *error policies*. These policies, which exist for transaction errors and for general errors, are declaratively applied to a shared object or its methods and can execute compensation code whenever an error occurs within that object or method.

For example, if concurrent modifications prevent a transaction context from being committed, this is usually because another peer is currently using the same objects. (In the Whiteboard example, this could be caused by two (or more) users simultaneously changing the size of a shape.) Therefore, the default transaction error policy waits for a configurable amount of time and then retries the operation.

In the Whiteboard application, it is not acceptable that any exception is thrown by the drawing object, simply because the application does not expect and handle it. Therefore, Listing 3 annotates the *SharedDrawing* class with the *UIErrorPolicy* attribute, which is defined to display a message dialog in case of an error, letting the user decide whether to retry the operation, ignore the failure, or abort the program. Note that the error handling mechanism is totally transparent to the user of *SharedDrawing* objects. Of course, more sophisticated types of error policies and compensation are equally possible.

4.3 Automatic Refresh

A shared object is accessible by multiple peers within a collaborative application. It will often be the case that an object's state is changed unbeknownst to most of the peers, and usually, it is necessary that some event be triggered in this case. For that, ATTRIBUTES &CO provides the concept of *auto-refreshed* shared objects. A shared object is declared to be auto-refreshed by marking its class (*AutoRefreshed* attribute, see Listing 3). Whenever the state of such an object is altered, the change is reported to all peers on the network.

Sometimes, it is not sufficient to just propagate a state change to all the peers; it must also trigger an update action. For instance, in the Whiteboard application, the drawing must be repainted whenever a peer adds, changes, or removes a shape. Therefore, in Listing 3, the *OnDrawingChanged* method is annotated with the *OnAutoRefresh* attribute; it will be called immediately after the shared object's state has been refreshed and invoke the *DrawingChanged* event.

With the introduction of automatic refresh, additional concurrency issues are introduced. It is not acceptable that a shared object's state (say, the number of shapes in the Whiteboard example) "magically" changes while a method of that object (e.g. *AddShape*) is being executed; the method must be able to rely on consistent state.[2] ATTRIBUTES &CO automatically ensures that a shared object's methods are protected from such multi-threading issues: refresh data is queued until all currently executing methods have exited.

For finer control, ATTRIBUTES &CO can also be instructed to acquire a lock on a special *monitor* object. If the lock is held while a refresh notification arrives, ATTRIBUTES &CO waits until the lock is released. ATTRIBUTES &CO guarantees that it holds or requests no additional locks while it keeps a lock to the monitor, so that no deadlocks are introduced by this mechanism.

In Listing 3, all methods use the default protection provided by ATTRIBUTES &CO.

4.4 Code Size Comparison

The listings provided in this paper could only give an excerpt from the source code of the Whiteboard application. Table 1 compares the lines of code of the full implementations of the Whiteboard's object model.

Table 1. Code size comparison (Whiteboard object model)

Class Kind	Implementation	Lines of Code	Code size increase
IDrawing implementation	Local	59	**0%**
	Shared, declarative	79	**34%**
	Shared, imperative	205	**247%**
Shape implementations	Local	189	**0%**
	Shared, declarative	196	**4%**
	Shared, imperative	237	**25%**

4.5 Current Implementation and Outlook

Currently, ATTRIBUTES &CO has been implemented as a prototype for the .NET 1.1 platform; prototype and production quality versions for Java 1.5 and .NET 2.0 are planned. The prototype is based on the CORSO middleware and uses a dynamic code generation strategy based on the *DynamicProxy* library [Verissimo, 2004]: whenever an object is created which is to be shared via the CORSO space, a dynamic proxy is created for it at runtime. The proxy wraps the original object and intercepts all method calls to it, deciding – based on the annotations declared on the object and method, and also considering the current state (e.g. existence of a transaction context, etc.) – whether middleware functionality must be invoked. DynamicProxy's support for *mixins*, which are objects that add additional functionality to the wrapped object instance, is used to add support for serializing the object's state and for refreshing and persisting it from and to the space. The object is also added to a global notification handler, which uses CORSO's notification functionality for the automatic refresh feature. Since the scope of this paper is not that of technical details, we will not further discuss the

[2] Note that this does not generally apply to *user interactions* within the application. Usually, a user interaction consists of many subsequent method calls, which are not necessarily executed in an atomic way. It is often acceptable (and even desirable) that automatic refresh events occur between method calls during such an interaction.

implementation of ATTRIBUTES &CO. For the interested, the source code of ATTRIBUTES &CO is available for download. [AttributesCo, 2005]

The existing implementation shows good performance and debugging experience (generated code is mapped back on the respective original source code), and serves as a successful proof of concept. In our future work, we will fully decouple the concerns of distribution from the language binding, allowing their implementation, application, and extension using *aspects* [Schmied and Kühn, 2004]. We will also investigate improvements for debugging and more complex transaction semantics.

5. CONCLUSION

In this paper, we introduced ATTRIBUTES &CO, which is a declarative language binding for VSM-based middleware systems for modern object-oriented languages based on extensible metadata. We provided a real-life collaborative example application, the distributed Whiteboard, to show the problems of imperative language bindings of intelligent middleware layers. We then showed the same application as implemented using ATTRIBUTES &CO, showing its simplicity and natural usage, and explaining its core principles: shared objects, state, business methods, error policies, and automatic refresh. By giving detailed numbers about the full Whiteboard object model implementations, we illustrated how the use of ATTRIBUTES &CO can lead to minimal code size, improved code quality, and thus better development productivity.

REFERENCES

AttributesCo, 2005. *Attributes &Co Prototype ProxiedCorso*.
[WWW Document] http://stud3.tuwien.ac.at/~e0327182/proxiedcorso.html [valid as of 08/07/2005].

Austin, C., 2004. *J2SE 5.0 in a Nutshell*.
[WWW Document] http://java.sun.com/developer/technicalArticles/releases/j2se15/ [accessed on 17/05/2005].

European Computer Manufacturers Assoc. (ECMA), 2002. *Common Language Infrastructure.* Standard ECMA-335.

Forman, I.R. and Forman, N., 2004. *Java Reflection in Action*. Manning Publications, Greenwich, USA.

Gray, J. and Reuter, A., 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Series in Data Management Systems, San Francisco, USA.

Kühn, e., 1994. Fault-Tolerance for Communicating Multidatabase Transactions. *Proceedings of the 27th Hawaii International Conference on System Sciences (HICSS).* Wailea, Maui, Hawaii, pp. 323-332.

Kühn, e., 2001. *Virtual Shared Memory for Distributed Architecture*. Nova Science Publishers, New York, USA.

Kühn, e., 2005. *eXtensible Virtual Shared Memory (XVSM) – A GRID Enabled Space Based Computing Architecture for Distributed Computing.* Technical Report, Vienna University of Technology, Institute of Computer Languages.

Kühn, e., Nozicka, G., 1998. Post-Client/Server Coordination Tools. *Coordination Technology for Collaborative Applications,* Springer Series Lecture Notes in Computer Science, Vol. 1364, pp. 231-253.

Lopes, C.V. and Kiczales, G., 1997. *D: A Language Framework for Distributed Programming*. PARC Technical Report SPL97-010 P9710047.

Lukosch, S., 2002. Adaptive and Transparent Data Distribution Support for Synchronous Groupware. *Groupware: Design, Implementation, and Use, 8th International Workshop, CRIWG 2002.* La Serena, Chile, pp. 255-274.

Schmied, F., 2003. *MEXlib – A Library for the Analysis, Manipulation, and Extension of .NET Metadata in Unmanaged Code.* University of Applied Sciences, Hagenberg, Austria.

Schmied, F. and Kühn, e., 2004. Distributed Peer-to-Peer Application Development with Declarative and Aspect-Oriented Techniques, *Conference Proceedings of International Symposium on Leveraging Applications of Formal Methods*. Paphos, Cyprus, pp. 150-157.

Sun Microsystems, 2003. *JavaSpaces™ Service Specification*.
[WWW Document] http://java.sun.com/products/jini/2.0/doc/specs/html/js-title.html [accessed on 19/05/2005].

Verissimo, H., 2004. *Castle's DynamicProxy for .NET*.
[WWW Document] http://www.codeproject.com/csharp/hamiltondynamicproxy.asp [accessed on 17/05/2005].

Wikipedia, The Free Encyclopedia, 2005. *Declarative programming*.
[WWW Document] http://en.wikipedia.org/wiki/Declarative_programming [accessed on 23/05/2005].