EVENT MODELS:

AN EVALUATION FRAMEWORK

Guðlaugur Stefán Egilsson

Submitted by Guðlaugur Stefán Egilsson
to the University of Skövde as a dissertation towards the
degree of M.Sc. by examination and dissertation in the
Department of Computer Science
(October 1999)

I certify that all material in this dissertation which is not my
own work has been identified and that no material is included
for which a degree has already been conferred upon me.

Signed ..............................................………….
# Table of Contents

1. INTRODUCTION .................................................................................................................. 1  
   1.1 AIMS ................................................................................................................................. 1  
   1.2 OBJECTIVES .................................................................................................................... 1  
   1.3 AREA OF CONTRIBUTION ............................................................................................ 1  
   1.4 OVERVIEW ...................................................................................................................... 1  

2. BACKGROUND ..................................................................................................................... 3  
   2.1 COMPONENTS ................................................................................................................. 3  
      2.1.1 Introduction ............................................................................................................... 3  
      2.1.2 History ...................................................................................................................... 4  
      2.1.3 Various definitions ................................................................................................... 5  
      2.1.4 Characteristics ......................................................................................................... 8  
   2.2 EVENTS AND EVENT MODELS ..................................................................................... 9  
      2.2.1 Introduction ............................................................................................................... 10  
      2.2.2 Callbacks .................................................................................................................. 11  
      COM Connection Points .................................................................................................. 13  
      2.2.4 The CORBA Event Service ...................................................................................... 14  
      2.2.5 Others ..................................................................................................................... 16  
      2.2.6 Advanced event services - Snoop ......................................................................... 18  
      2.2.7 Terminology ............................................................................................................ 21  

3. PROBLEM DEFINITION ...................................................................................................... 22  
   3.1 MOTIVATION .................................................................................................................. 22  
   3.2 PROBLEM STATEMENT ................................................................................................. 23  

4. EVALUATION FACTORS ................................................................................................... 24  
   4.1 INTRODUCTION .............................................................................................................. 24  
   4.2 EVALUATION CRITERIA ................................................................................................. 26
4.2.1 Contracts .................................................................................................................. 26
4.2.2 Polymorphism .......................................................................................................... 29
4.2.3 Encapsulation .......................................................................................................... 31
4.2.4 Concurrency ............................................................................................................ 33
4.2.5 Semantics ................................................................................................................ 37
4.2.6 Safety ....................................................................................................................... 40
4.2.7 Service level ............................................................................................................ 43
4.2.8 Heterogeneity .......................................................................................................... 46

5. EVALUATION OF THE CORBA EVENT SERVICE AND THE COM
ICONNECTIONPOINT EVENT MODEL ............................................................................ 49

5.1 INTRODUCTION ......................................................................................................... 49

5.2 EVALUATION ............................................................................................................. 50

5.2.1 Contracts ................................................................................................................ 50
5.2.2 Polymorphism ......................................................................................................... 54
5.2.3 Encapsulation .......................................................................................................... 56
5.2.4 Concurrency ............................................................................................................ 58
5.2.5 Semantics ................................................................................................................ 60
5.2.6 Safety ....................................................................................................................... 63
5.2.7 Service level ............................................................................................................ 65
5.2.8 Heterogeneity .......................................................................................................... 67

6. RESULTS ...................................................................................................................... 71

7. CONCLUSIONS AND FUTURE WORK ..................................................................... 74

7.1 CONCLUSIONS .......................................................................................................... 74

7.2 LIMITATIONS ............................................................................................................ 74

7.3 FUTURE WORK .......................................................................................................... 75

8. ACKNOWLEDGEMENTS ............................................................................................. 76
9. GLOSSARY ........................................................................................................77

10. BIBLIOGRAPHY AND REFERENCES .......................................................... 78

11. APPENDIX A. OVERVIEW OF THE EVALUATION FRAMEWORK .............. 82

12. APPENDIX B. SELECTED LISTINGS .............................................................. 84

  12.1 LISTING 1. A SAMPLE COM INTERFACE FOR PASSING EVENTS ................ 84

  12.2 LISTING 2. THE PROCESS OF USING COM DISPATCH INTERFACES FOR PASSING EVENTS .... 85
### Table of figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Observer Pattern. Adapted from [Gamma95]</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>A sample connection points setup. Adapted from [Rogerson97] with some</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>generalizations.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Event channel setup</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>Overview of the Java Event Model [JavaBeans].</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>Snoop event classification [Cha93]</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>Snoop event operators [Cha93]</td>
<td>19</td>
</tr>
<tr>
<td>7</td>
<td>Event occurrences</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>Domain space</td>
<td>25</td>
</tr>
<tr>
<td>9,</td>
<td>Inner workings of the STA model</td>
<td>35</td>
</tr>
<tr>
<td>10</td>
<td>Problem of asynchrony</td>
<td>38</td>
</tr>
<tr>
<td>11</td>
<td>Classification of Failure Models (adapted from [Hadzilacos93])</td>
<td>41</td>
</tr>
</tbody>
</table>
Abstract

Event based programming is an important metaphor used in a variety of applications. Research and practice in this field comes primarily from two distinct sources, component software and databases. In both those fields, the need for asynchronous notifications has led to the specification and implementation of event models, but with somewhat different emphasis.

This dissertation defines an evaluation framework for evaluating event models. In doing so, it defines several factors that are important when reviewing different event models with respect to implementing applications or components that require event notification mechanisms.

It has been suggested that the event models defined for COM and CORBA can each be used as the basis for implementing advanced event services. The framework presented in this dissertation is used to evaluate these two event models with respect to their capability to support an advanced event service originating from active database research.
1. Introduction

1.1 Aims

The aims set for this dissertation are the following:

To establish a base for evaluating component based event models. To investigate how “advanced” event applications defined in the literature relate to event model approaches found in commercial component models.

Hypothesis: The event models of current component models need to be improved to support advanced event applications.

1.2 Objectives

To fulfill the aims described above, the following will be performed: Establish a view on events in software, what are they, and more importantly, what they are not. Define what a software component is, and the requirements that a piece of software has to fulfill to be called a component. Create an evaluation framework for event models. Then use the framework to evaluate the suitability of the event models in COM and CORBA for detecting composite events, as a sample advanced event application, in an attempt to falsify the hypothesis.

1.3 Area of contribution

This dissertation contributes to the areas of event based programming and component software.

1.4 Overview

In chapter 2, the background for this work in the realm of components and event
models is established. Chapter 3 contains the problem definition where the motivation for this work is given, along with a problem statement. Chapter 4 introduces the evaluation framework developed, and Chapter 5 presents evaluations of the COM and CORBA event models in terms of Snoop-style complex event detection, based on the framework presented in Chapter 4. Chapter 6 summarizes the results from the evaluations in Chapter 5, while Chapter 7 presents conclusions and future work. Appendix A provides a short overview of the framework which is useful for establishing the scope of this work. Appendix B contains a couple of selected listings.
2. Background

2.1 Components

2.1.1 Introduction

Component technology has been The Holy Grail of software engineering for 30 years. Predictions that component technology would end the so-called software crisis appeared as early as 1969 [Szyperski98]. The quest for component technology is by no means coincidental, all mature engineering disciplines have adopted components as a means to distribute expertise and increase productivity. Several motivations for seeking component technology have been identified, not all of them technical. Market and sociological issues are amongst the driving motives, and are probably just as important as the technical ones.

The single most important reason for the adoption of component technology is probably the constant need for increased reuse. Reuse of effort already put into developing applications is seen as one of the most effective ways to lower the cost of developing new applications, and allow software developers to reach higher levels of functionality. Although reuse can also be achieved by other means, such as by using class libraries and application frameworks, it is the combination with other factors that make components important.

In chapter 2.1.2 a brief history of components will be presented, followed by a detailed review of component research in the literature in chapter 2.1.3. A summary of component characteristics is then presented in chapter 2.1.4.
2.1.2 History

The idea of components is certainly not new. Microsoft introduced the first truly successful component architecture in 1992, the Visual Basic Controls (VBX) architecture with its Visual Basic development tool. Although it was not specifically designed to allow third parties to create components, it spawned a substantial market of third party components ranging from custom edit controls and spreadsheets to imaging editing over to instrument controlling components [Udell94]. However, older still are modern operating systems. Operating systems can be viewed as component architectures [Szyperski98], allowing course-grained components (applications) to interact using the file system and inter-process communication primitives as its “wiring-standard”. Applications providing plug-in functionality can also be considered as defining a component model, which is then usually streamlined for the application’s domain. Examples of such architectures are Netscape Navigator and Apple’s QuickTime [Szyperski98].

Since the success of the Visual Basic model, more component-based approaches have emerged. Borland International (now known as Inprise Corporation) introduced Delphi in 1994, which defined a developer-centric component model, the Visual Component Library (VCL). The VCL was based on proprietary object files that were linked into the target application, but has since then evolved into a more open format, also supporting C++. Microsoft refined its component strategy, first around OLE and OCX’s, which later morphed into COM and ActiveX, which is the base for Microsoft’s component strategy today. From a more corporate oriented environment, OMG came forth with the Common Object Request Broker Architecture (CORBA) standard which is commonly regarded as one of the major players in the component world. Finally Sun Microsystems
have made a large impact on the software industry in recent years with its Java language/platform. As the component model of Java, Sun has developed JavaBeans which defines various mechanisms necessary for components. The three last mentioned component models (COM, CORBA and JavaBeans) are commonly regarded as the most important ones on the market today.

2.1.3 Various definitions

Within the literature, it is not very clear what is meant by the term “software component”, although there appears to be a consensus that they are for composition [Szyperski98][Murer97]. Several attempts have been made to define what a software component is, but so far no one definition has been accepted as “the definition”. Below are listed a few definitions that have been suggested. At the end of this chapter, the definition of component used by this dissertation will be given, which will hopefully make it clear what is meant by the term “software component”.

Robert Orfali, Dan Harkey and Jeri Edwards [Orfali96] (detailed descriptions omitted for brevity):

A “minimalist” component:
- is a marketable entity.
- is not a complete application.
- is usable in unpredicted combinations.
- has a well-specified interface.
- is an interoperable object.
- is an extended object.

And
A “supercomponent” adds support for:
- security,
- licensing,
- versioning,
- life-cycle management,
- support for visual assembly,
- event notification,
- configuration and property management,
- scripting,
- metadata and introspection,
- transaction control and locking,
- persistence,
- relationships,
- ease of use,
- self-testing,
- semantic messaging,
- self-installation.

In an IDC white paper [Garone98], a component is defined as a piece of software that

- Is both discrete and well defined in terms of its functionality
- Provides standardized, clear, and usable interfaces to its methods
- Can run in a container, with other components, or standalone (or any combination of these)

T. Spitzer [Spitzer97], defines component based development as

... the practice of delivering solutions by building or buying interoperable components. Components’ ability to interoperate results from their adherence to a widely accepted software infrastructure that defines a common mechanism for such bundles of functionality to work together within a common container.

In his thesis, L. Deri [Deri97] defines a component as a

Static abstraction with bi-directional plugs

Static emphasizes the fact that a component has a long lifetime and is seldom changed. Abstract means it is encapsulated with a well-defined boundary. Bi-directional means that a component should be capable of two-way communication in a peer-to-peer manner rather than the one-way communication of client-server models.

Allen [Allen98] defines components a little differently.
A component is an executable unit of code that provides physical black-box encapsulation of related services. Its services can only be accessed through a consistent, published interface that includes an interaction standard. A component must be capable of being connected to other components (through a communications interface) to form a larger group.

Allen continues to define what he means by a service:

A service is a type of operation, that has a published specification of interface and behavior, involving a contract between the provider of the capability and the potential consumers. Using the service description, an arms length deal can be struck that allows the consumer to access the capability.

This definition is a little more restrictive. Under this definition, neither source code libraries nor object libraries qualify as they are not executable¹ and do not provide black-box encapsulation of related services. Standard DLLs² do not qualify either, as they do not adhere to an interaction standard. However, objects that publish their operations and data through OMG’s CORBA or Microsoft’s (D)COM interfaces, or comparable architectures do qualify as components under this definition.

These definitions given above circumvent the relation of components to object-orientation for the most part. It is established that components do not have to be created with object oriented languages [Orfali96][Udell94][Spitzer97][Szyperski98], something best illustrated by the VBX architecture, the main implementation language for components being plain (and not so simple) C.

In [Booch87], Grady Booch defines software components as follows:

---

¹ In this context, “executable” means compiled code (i.e. not source code or object code), e.g. in the form of dynamic link libraries.
² Note that COM objects, as well as the old VBX standard, use the DLL format as their binary standard. What is different from standard DLLs, is that they provide standardized entry points.
Component: A container for expressing abstractions of data structures and algorithms; a component should be a logically cohesive, loosely coupled module that denotes a single abstraction characterized as an abstract state machine or an abstract data type.

By this definition, independent deployability is not required.

Notice that inheritance appears in only one of these definitions above. One of the reasons for inheritance not being a fundamental property of component models is that inheritance breaks encapsulation [Babtista96][Weck96][Szyperski98], which is one of the fundamentals of component models (see definitions above). Another reason is that the opposite has already been established (that a component model can be successful without inheritance), and has not been refuted.

Murer [Murer97] gives the definition agreed upon at WCOP’96:

Definition 1- Component

A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A component can be deployed independently and is subject to composition by third parties.

This is also the definition adhered to in this dissertation, as well as by Clemens Szyperski [Szyperski98] which is this dissertation’s primary source on components.

2.1.4 Characteristics

Although distribution, objects and components are really orthogonal concepts [Szyperski98], the characteristics of those concepts are often mixed when components are being discussed. Here the relevant characteristics of components that result from the

3 WCOP’96 = International Workshop on Component-Oriented Programming (in conjunction with ECOOP’96)
definition given above, or have been mentioned elsewhere in the literature, are explained.

Components are units of composition. This requires that all components adhering to a certain component model have to be polymorphic at the basic level, i.e. all components must have some basic entry points as the component model has be able to put the component in its place so it can be activated. An operating system does this by requiring all executables to have a common entry point, executable files are generally not usable across operating systems, even if they run on the same processor. Component models such as COM and CORBA do this by requiring components to be registered in a registry/repository, along with supported entry points (interfaces).

Components are independent. This can be seen to some degree in all of the definitions listed above, and is one of the characteristics on which there is broad agreement. This is not so surprising, as independence is a direct result of the primary requirement for components, namely that they are for composition. Independence of components would, however, mean little if they would all have to originate at the same source. Independent production, the fact that components originate at independent sources and are integrated by third parties, is the source of most technical advantages and problems in component technology [Szyperski98]. The requirement of independent deployment is obviously a direct consequence of independent production, since independently produced components are useless unless they can be brought together for composition.

2.2 Events and event models

Event:
2 a: something that happens: occurrence b: a noteworthy happening c: a social occasion or activity [MW].
2.2.1 Introduction

*Event driven programming is a technique used in software that operates a program in response to events as they occur.*

*Lewis98*

**Definition 2**

*An event is an atomic (happens completely or not at all) occurrence.*

*Cha93*

This dissertation adheres to this definition of an event. However, any part of any computation may fit that definition, and it is therefore too broad to be of use when discussing event models in the context of software. Another issue to be dealt with when considering events, is the mechanism used to deliver events, or event notifications. As the author has not come across a formal attempt to define what an event notification is in a software engineering sense, the subject will be tackled here.

An event type is defined by the source of a potential event. The classic example is a component encapsulating a button in a windowing system. The component would typically expose one event source (event type) that would notify event consumers when the button is clicked, and another source for notifying when it is double-clicked. By subscribing to the “on-click” event, a client is subsequently notified whenever the button is clicked, without requiring any modifications to the component encapsulating the button class. Other examples of events, from the database world, include the beginning of transactions, begin-of-insert, end-of-insert, etc. [Cha93]

The event connection is set up by the sink, or consumer of a potential event. This is the main difference between regular method calls, where the caller decides who the callees are, and event calls, where the callee decides who the caller is. An event call is a
service the caller is providing to the callee, which is a reversal of the normal role, where the callee is performing a service for the caller. Normal method calls are to request a service from the callee, while in this case it is to provide a service that has been requested earlier [Szyperski98].

As a result, the following definition can be formed:

**Definition 3 - Event notification**

*An event notification is an implicit invocation, whose target is specified by a subscriber. Such an invocation is a service of the notifying component.*

It follows that the caller does not depend on such calls being made for completing its function. It may also be subject to any level of indirection, such as is required by remote calls.

**Definition 4 - Event model**

*An event model defines mechanisms for establishing and discontinuing event communications. It may also define how events are signaled and their parameters computed and packaged.*

### 2.2.2 Callbacks

Definitely the most common model for providing event notification, is the technique of callbacks. Callbacks depend on a feature commonly found in programming languages, such as C++ and Pascal, i.e. the possibility to declare and set *method variables*, and later call a method pointed to by the variable. In the context of events, a method variable is essentially an event type, and each call to the method is an event notification. The key point here is that the component that calls some method using a callback variable, does not know the conceptual destination of the call, all it knows is the address of the method,
that the destination conforms to the calling convention defined by the method variable and that it wants to be called under the prescribed circumstances.

More elaborate versions of callbacks make use of object or interface variables rather than direct method variables, adding a level of indirection. The object or the interface that is put as a callback variable then offers a method, or possibly a set of methods, that the callback owner can use to notify the object. A classic example of this is the observer pattern [Gamma95].

The intent of the observer pattern is to “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.” Key methods in the pattern from our point of view, are the Attach(Observer), Detach(Observer) and Update(). Attach adds an observer to a list of observers, i.e. sets it as a callback variable. Detach is used to remove it from the list. Update() is the callback method itself. Update denotes that an Observer should get a new state from its observed object, i.e. the subject.

Figure 1 Observer Pattern. Adapted from [Gamma95]
With variants, the callback model is used for event notification by all component models mentioned in this dissertation. Event models have been described as a generalization of the observer pattern [Szyperski98]. Delphi uses pure callbacks as its event model, as described in section 2.2.5.

2.2.3 COM Connection Points

The primary approach of COM to events involves the so-called “Connectable Objects”, or connection points, which will be described very briefly here. Connection points are designed to provide support for enumerating event types and subscribing to them through a standard set of interfaces. The interfaces defined for connection points are four in all. These interfaces are IConnectionPointContainer, IEnumConnectionPoints, IConnectionPoint and IEnumConnections.

IConnectionPointContainer defines two functions, FindConnectionPoint and EnumConnectionPoints. EnumConnectionPoints returns an IEnumConnectionPoints interface that can be used, as the name suggests, to enumerate all connection points supported by the object implementing IConnectionPointContainer. When an

Figure 2 A sample connection points setup. Adapted from [Rogerson97] with some generalizations.
IConnectionPoint interface has been obtained, it can be used to subscribe/unsubscribe to the event managed by the connection point, or in COM terms, advise/unadvise the event source of the event sink. The ConnectionPoint also provides the capability to enumerate all active subscriptions using EnumConnections.

Notice that none of the interfaces defined for connection points is for defining the event interface itself. The IEventSink interface shown in Figure 2 is just a sample name put in for explanation purposes. This has the consequence that event interfaces in COM are generally not compatible with each other.

For a further reading on connection points see [Rogerson97] and [Eddon98].

2.2.4 The CORBA Event Service

One of the services specified in the CORBA services specification [CORBAser] is the CORBA Event Service. As the name suggests, the specification defines what the event model of CORBA applications should look like. The specification is built around three main concepts; event consumers, event suppliers and event channels, and two communication models, the pull and push models. To complicate things a little more, it defines both typed and untyped (generic) event communications. To simplify this background, only the generic push model will be presented here.

To begin with, the CORBA Event Service defines two generic interfaces that are used for event communications, the PushSupplier and PushConsumer interfaces. They can be thought of as plugs; when you have a PushSupplier on one end, you must have a PushConsumer on the other. These interfaces provide the basic capabilities of communicating events using the Any type, and to disconnect from each other. They do not provide the capability to connect; that is left to higher level interfaces.

To support further decoupling of suppliers and consumers and to enable multicast, the
EventChannel interface is introduced. An event channel acts as a mediator between suppliers and consumers, fully decoupling them from each other. To support this role, some additional interfaces are defined. First of all, an event channel supports the EventChannel interface, which allows clients of the channel to destroy it and to acquire the ConsumerAdmin and SupplierAdmin interfaces. The admin interfaces are designed to allow consumers and suppliers, respectively, to connect to the event channel. The admin interfaces perform this duty by returning proxy consumers/suppliers implemented by the ProxyPushSupplier/ProxyPushConsumer interfaces. Those interfaces are descendants of the respective PushSupplier/PushConsumer interfaces, but add the capability to connect, that was lacking before. The process of setting up an event channel and connecting to it is then something like this (in pascal-like pseudocode without any error checking).

```pascal
EventC := GetEventChannel;
EventC.SupplierAdmin.obtain_push_consumer.connect_push_supplier(PushSupplier1)
EventC.ConsumerAdmin.obtain_push_supplier.connect_push_consumer(PushConsumer1)
```

![Figure 3 Event channel setup](image-url)
For an illustration of the event channel setup, see Figure 3

2.2.5 Others

Other component models also define how event communications are performed. Here I will describe two of them briefly, the one used by Java, and the one used by Borland Delphi, which is actually the model that originally inspired this work.

Java

In the original JDK 1.0, events were solely designed with GUI applications in mind, but were replaced in JDK 1.1 by a more generic class hierarchy which also supports non-GUI events.

*Conceptually, events are a mechanism for propagating state change notifications between a source object and one or more target listener objects.* [JavaBeans]

The current event model in Java is built around those concepts. When an event occurs, an event object derived from EventObject is constructed and passed to interested EventListeners which have registered with the event source. Thus, the event model supports multicasting as multiple listeners can generally be registered with each event source, although an event source might decide to support only single-casting, denying subscription requests when it already has one subscriber. For each concrete event class, an event listener interface is defined which an interested class must implement to be able to receive that type of event. As an example, if a class wants to receive events of type FooEvent, it implements the FooListener interface. The event model is designed so that advanced event services can be built for it.

The JavaBeans specification [JavaBeans] suggests that a generic demultiplexing event adapter can be built to simplify event management when an event listener must listen to
the same event from multiple sources. Event sources can be queried for their available events using the Java introspection mechanisms.

Borland Delphi

Delphi is one of the early development tools that made heavy use of components. Although what is called a component in Delphi would not qualify as a component\(^4\) under the definition this dissertation adheres to, the basic philosophy that existing “components” cannot be modified, along with the requirement of visual object editing results in at least one common requirement, namely that there should be an event model present. Delphi goes the way of callbacks for event notification. This leads to a highly efficient mechanism with practically the same performance as regular calls, where the

---

\(^4\) From version 3, Borland Delphi has supported the notion of “packages”, which are more on the lines of components as they are defined here.
only action necessary for subscribing to an event is to set a callback variable. The only extra action necessary for firing an event is to check whether the callback variable is set to a non-nil value. To support more advanced applications such as persistence of event callback assignments and automatic code generation, the Delphi compiler includes extensive run-time type information (RTTI) on event callbacks.

Although highly efficient, there are at least two serious drawbacks to this method. One is that it does not support multicasting; each object can only accept one subscriber to each of its events at any given time. Another is that when an event handler is defined, it has to match the method signature of the event callback, which has the consequence that the same event handler cannot handle events with different method signatures. In other words, events in Delphi are not polymorphic.

2.2.6 Advanced event services - Snoop

In addition to the basic event services defined by the component models described here, several “advanced” event services have been suggested in the literature. In the CORBA Event Service specification, OMG [CORBAser] suggest complex event detection, event filtering and event persistence as potential applications for CORBA Event Channels.
One particularly fertile ground for research on events and their applications has been in the field of active databases. Research in this area gave rise to the Event-Condition-Action (ECA) rule concept, and in particular it lead to the introduction of complex events, here inspired by Snoop, “An Expressive Event Specification Language for Active Databases” [Cha93]. Snoop defines a language for specifying complex events, and outlines a design for an event service to support the language. It also defines some fundamental underlying concepts, like the notion of an “event” (see definition 2).

![Figure 5 Snoop event classification [Cha93]](image)

![Figure 6 Snoop event operators [Cha93]](image)
The idea of complex events, also known as composite events, is to compose primitive events using event operators, creating a new event type. A simple example of a composite event is “a change of temperature AND a change of pressure”, denoted T1 and P1 in Figure 7, which might trigger a reevaluation of environment conditions. A classification of events and event operators can be seen in Figure 5 and Figure 6. The event classification illustrated in is made with relational databases in mind, but the design is application and model independent in principle [Cha93].

When composing events in this manner, the problem of event consumption (or parameter context) becomes an issue. When using single events to trigger conditions/actions, this is a non-issue, as primitive events are atomic and there is no possibility of accumulation. Consider the example before, “a change of temperature AND a change of pressure”. If we assume that the pressure is fluctuating all the time, but there is only an occasional change in temperature, the question arises of how to handle all the event notifications for pressure change (p1a – p1c) before the one temperature change notification (t1a) that caused the composite event to be detected. For this purpose, Snoop defines four parameter contexts; recent, chronic, cumulative and continuous. The simplest of these, and the most intuitive, is the recent context, which only takes the most recent event occurrences into account, i.e. in the aforementioned example, only the last temperature and pressure changes (p1c and t1a) would be passed on to a subscriber.

```
P1   p1a   p1b   p1c
T1   |     |     |     t1a
      0     |     |     |     time
```

**Figure 7 Event occurrences**
2.2.7 Terminology

The terminology for events can sometimes be a little bit confusing, especially when discussing different specifications, which each uses its own terminology. To ease things a little, the terminology is briefly explained here.

<table>
<thead>
<tr>
<th>Terminology</th>
<th>Term for origin</th>
<th>Term for delivery contract</th>
<th>Term for event instance</th>
<th>Term for destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>COM</td>
<td>Connection point</td>
<td>Event interface</td>
<td>Event call</td>
<td>Event sink</td>
</tr>
<tr>
<td>CORBA</td>
<td>Event supplier</td>
<td>Event interface</td>
<td>Event call</td>
<td>Event consumer</td>
</tr>
<tr>
<td>Delphi</td>
<td>Event property</td>
<td>Event method type Listener interface</td>
<td>Event call</td>
<td>Event handler</td>
</tr>
<tr>
<td>Java</td>
<td>Event source</td>
<td>Event object</td>
<td>Event listener</td>
<td></td>
</tr>
</tbody>
</table>

The respective terminology will be used whenever the discussion is centered on that model. In general discussion, any of those terminologies can be used.

A frequent source of misunderstanding in the discussion on events and event detection, is the lack of differentiation between event types and event instances, both of which are frequently referred to simply as events. This is very similar to the common class/object mix-up. This has been avoided in this dissertation, and where the term “event” is used standalone, it is not imperative to distinguish between the two.
3. Problem definition

3.1 Motivation

Large classes of software applications are reactive in their nature. User interface programming is about reacting to user-generated events and real time applications are about reacting to input generated by environmental sensors [Eriksson98]. Database applications do benefit from reactive mechanisms and a significant amount of research has been devoted to (re)active databases.

The trend in application development has been moving towards component models, such as Microsoft’s Component Object Model, (COM), and OMG’s Common Object Request Broker Architecture (CORBA). Component oriented programming relies largely on the practice of assembling components from different sources. Such components are frequently not under source code control of the user, and have to be considered as being “black-box“ components. Mechanisms that provide the user of such components with the ability to detect and react to events occurring inside the components are obviously important, as the alternative is to use potentially frequent polling, wasting computing resources.

Thus, reactive programming techniques are important to implement applications that are reactive in their nature, and even more importantly, reactive programming techniques have been identified as one of the decisive factors that enables component oriented programming. All component models provide event notification models “as one particularly flexible form of component instance assembly.”[Szyperski98]. However, most of them have not been designed with advanced event applications in mind.
Considerable research has been focused on reactive mechanisms in databases, suggesting advanced applications of events that can be beneficial in several application domains. These include event filtering, event logging (persistence) and complex event detection. It has also been suggested that such services would be useful in non-database applications [Le98][Gatziu97]. The feasibility of using existing infrastructures to support such services is however not well known. This is the focus of this dissertation.

### 3.2 Problem statement

This dissertation addresses the fundamental properties of event models in the broadest sense. By surveying literature on component programming, concepts that influence component event models are compiled into a list of factors to be used for evaluating different event models. In particular, factors that are necessary to consider for supporting advanced event services are covered. To validate the relevance of these factors, an evaluation of two well specified event models is performed in the terms of a well known advanced application of events, namely complex event detection.
4. Evaluation factors

4.1 Introduction

This chapter lists factors that are relevant to component construction, and influence the event model employed by the component model. The list of factors was developed by creating a list of candidate technical concepts found in literature on components and eliminating the least relevant, where there was no or minimal motivation for inclusion. This method excludes factors such as distribution, security, usability and market issues.

Each factor is described in a subchapter, which again has three sub-topics:

- **Description** – A brief description of the concept, and especially its relevance to component software. This is often based on material introduced in the background chapter, but may also stand on its own.
- **Motivation** – Why does this factor influence the event model, i.e. the justification for including it in the base for evaluation.
- **Criteria** – What are the attributes that need to be evaluated when considering this factor of event models.

In many cases, component models are designed to be as domain generic as possible, allowing for extensions that make them more domain-specific. This is the approach of COM; the fundamental technology underlying COM (i.e. IID$s$, vtbl$s$, IUnknown) is very generic in nature. By adding certain extensions, the model will become more domain specific. For example, IOleInPlaceObject is one of the interfaces extending COM into the user interface domain. Figure 8 illustrates how models vary from the domain generic to the domain specific. Examples of such a variation are the domains of database applications on one hand, and real time applications on the other. If we consider the
requirements on a component model for the database and real time domains, we get a requirement for transaction support and good average performance in database applications. In real time applications, we get a requirement for contingency plans and predictability, amongst others.

In the context of events, the question here is the extent to which an event model supports different connection modes required by the application. Different application domains make different demands on the event model. As an example, in the case of user interface programming, multicasting is rarely needed and is therefore an undesirable luxury that does nothing but add runtime overhead. In the case of distributed database applications, the scenario is different. Database applications typically have multiple clients that are interested in similar events, and thus multicasting is a desirable feature to be found in the event model.

When choosing a component model for an application, it is important that it provides the appropriate amount of support for the application being built. You neither want too
much (too much overhead) nor too little (have to do too much yourself).

4.2 Evaluation criteria

4.2.1 Contracts

Description

A contract of a component is the definition of its behavior, both in terms of semantics and syntax. Contracts can be manifest in many ways, the most precise being the actual implementation. It is, however, desirable to have contracts defined on a higher abstraction level for many reasons, one being that the implementation does not suffice as a specification when multiple implementations exist (which implementation should be the specification?). Traditional ways of defining contracts include pre- and post-conditions (the Hoare triple), but more formal methods such as permissible histories [Szyperski98] are also known. An interface can be viewed as a self-standing contract between a provider of a service (i.e. a component) and its consumer [Szyperski98]. Such contracts can potentially be used by any number of providers, such as multiple components implementing the same interface and therefore fulfilling the same contract, and any number of clients. In such a case, the contract effectively decouples the consumer (client) from the provider (server), as the interface contract becomes the definition of how the participating parties communicate. An important part of any contract comprises the types of parameters and return values.

An important issue in contract specifications is versioning. Versioning is essentially the practice of keeping track of changes. In other words, versioning is about keeping track

---

5 See Delphi’s VCL with it’s non-multicast call-back mechanism as its event model as an example of a
of the evolution of programs. Commercial software versioning, as known to most people, e.g. after MS Word version 2, there came version 6, is rather superficial. The new version number states only that version 6 comes after version 2, and can therefore be expected to be an improvement in some sense (although some would argue that this example does not represent any improvement). In a component context, this notion of versions is not adequate. Components evolve independently, and if they are to continue to work properly together, there has to be a way to keep track of which versions of a component are compatible. This gives rise to the requirement of controlling versions of components, i.e. a mechanism that can determine whether a given component is suitable for the needs of a client.

Rogerson [Rogerson97] recommends that a new version (or as he puts it in COM terms, a new interface) should be declared as soon as one of the following change:

- Number of functions in an interface
- Order of functions in an interface
- Number of parameters in a function
- Order of parameters in a function
- Types of parameters in a function
- Possible return values from a function
- Types of return values from a function
- Meanings of parameters in a function
- Meanings of functions in an interface

As a more general rule, any change of the usage or implementation of an interface that can break the provider or client (i.e. cause any undesirable behavior), requires a new version of that interface. Of course, this also applies to event interfaces, as they also conform to an interface contract.

---

successful event model that does not support multicasting.
Motivation

Events are a part of component interfaces, and as such define a part of the contract of a component. The event model defines a part of the contract for an event, and thus how much of the contract specification is left to the interface designer. Events are in some cases defined by whole interfaces, which then need to have a well defined contract. The event model is therefore an important issue when an event interface contract is defined. The event model may specify how events are delivered (event delivery semantics), the availability of data types, impose a requirement to implement certain interfaces that are needed to support events, such as for subscription/unsubscription, or the calling convention used for event notification.

An event notification is a part of what dictates a component version change. If the parameters of an event change, or the conditions that trigger it, the semantics/syntax of a component change, which again requires a version change. This is obvious in the case of, let’s say, a change in parameter types. Such a change will in most cases make an event consumer completely incompatible with its intended source, resulting in a compile time or a run-time error when the source is bound to the consumer, or at latest when the event is fired. Incompatibilities may be more subtle when the firing conditions are changed, as such a change will only appear in the form of incorrect behavior, as the event will not occur when the client expected it to.

It is therefore clear that versioning is necessary to know whether an event source is appropriate for a given event consumer.

Criteria

Event delivery semantics contract
4.2.2 Polymorphism

: the quality or state of being able to assume different forms: as a:
existence of a species in several forms independent of the variations of sex b:
: the property of crystallizing in two or more forms with distinct structure

Description

Polymorphism: The ability of something to appear in multiple forms,
depending on context; the ability of different things to appear the same in a
certain context. [Szyperski98]

Polymorphism is by most authorities considered a cornerstone of object-oriented
programming. Rumbaugh [Rumbaugh91] for one mentions it as one of four defining
properties of object-orientation and Gamma et al [Gamma95] refer to polymorphism as a
key concept in object-oriented systems. Polymorphism is the ability to treat objects or
components of different types in the same manner [Rogerson97]. In an object oriented
context, this means that the same operation can be applied to different objects, with
different results. This requires different methods implementing the same polymorphic
operation to have the same method signature; that is, to have the same number and types
of parameters and return values [Rumbaugh91][Szyperski98]. It is possible to relax this
strict requirement by introducing covariance and contravariance, [Szyperski98]
meaning, respectively, that type preconditions (parameter types) are relaxed and type
postconditions (return types) tightened in subclasses. This capability is, however, rarely
supported, very few languages support covariance or contravariance [Szyperski98].
An alternative to using statically defined method signatures is to use method dispatch, also known as dynamic invocation. Method dispatch is built around the idea of packaging parameters for a method call into a commonly known structure, such as a list, and then passing that list of parameters as a parameter to a standard method call. The drawback of using such mechanisms is that they are inherently complex to use unless the language supports them. It is a technique frequently used by interpreted and scripting languages such as Visual Basic [Rogerson97][Eddon98]. They are also slower than regular method calls, as parameters have to be packaged/unpackaged at both ends, and the call has to be routed to the receiver. On the other hand, the power of this method is that it can be used polymorphically [Rogerson97], it is relatively easy to create a client that can call any interface that supports method dispatching in this manner. It is equally easy to create a component that can receive any method dispatched in this manner.

In other words, polymorphic type systems are commonly found in modern programming languages, in particular all object-oriented languages. Programming languages however, rarely support polymorphic methods, but that restriction can be circumvented by using method dispatch/dynamic invocation.

**Motivation**

Polymorphism is a very important concept for components. As generally agreed, components are for composition, and that requires participating components to be polymorphic at some level. By examining the Composite design pattern [Gamma95], this becomes evident. As an example, it is possible to write a piece of code that can work with any COM component, as they are polymorphic at the basic level, that is, they all implement one common interface, and can therefore all be manipulated using this interface (IUnknown).
The same cannot be said generally about events. First of all, events have been and still are implemented in several ways, as described in the background. Some of them are polymorphic in their nature while others are not. Some are somewhere in-between. The COM event model, for instance, is browsable as all events (or connection points as they are called in COM) are defined using a standard interface (IConnectionPoint). Such events won’t, however, accept subscriptions from everyone; a subscriber has to conform to a specific interface, i.e. implement specific methods whose signatures are dictated by the component implementing the IConnectionPoint [Rogerson97]. This makes it very difficult or even impossible to implement generic services such as event filters, event loggers and composite event detection for existing events, without being forced to create special adapters for each and every event type that needs to be supported.

**Criteria**

**Polymorphic event notifications**

**4.2.3 Encapsulation**

1: to enclose in or as if in a capsule <a pilot encapsulated in the cockpit>

2: epitomize, summarize <encapsulate an era in an aphorism>

intransitive senses: to become encapsulated

[MW]

**Description**

Encapsulation (sometimes referred to as information hiding in the literature) is about isolating the external aspects of an object from its implementation details [Rumbaugh91]. It is one of the fundamental techniques used to keep programs manageable, as it decreases the probability of changes of one object’s implementation having ripple effects throughout the program. This is one of the underlying themes of object technology
[Rumbaugh91], and is one of the many object technology concepts that are in common with component technology.

One of the characteristic properties of components is the following:

\[
A \text{ component is a unit of independent deployment.} \\
[Szyperski98]
\]

This has the implication that a component has to encapsulate all its constituent functionality, as otherwise it would not be independently deployable [Szyperski98]. Thus, encapsulation is one of the cornerstones of component technology, as without it this characteristic would not be achievable.

**Motivation**

Event models provide different levels of encapsulation for events. Event types are usually encapsulated by a component, hiding from external objects the actual points in its code defining the occurrence of events. This is important for the same reason that encapsulation of other information is important; in the absence of encapsulation a client of a component may start to depend on the actual occurrences of events rather than the logical ones provided by the component. One case of this scenario would occur in the case of a composite component (i.e. a component composed of other components) that publishes some of the inner components events. If the inner components are not properly encapsulated, clients might depend on the composed component’s events instead of the published events of the composite component.

Additionally, if event instances are first class objects, such objects need to encapsulate event data (parameters) to ensure consistent reception by all parties, as otherwise it would be possible to change event data at one recipient, leading to an inconsistent event
delivery at subsequent recipients. Similar results may occur by careless use of in/out parameters in event notifications.

**Criteria**

- Event source encapsulation
- Event parameter encapsulation

### 4.2.4 Concurrency

**Description**

Modern component models generally support concurrency, including COM, CORBA and JavaBeans, even on multiple levels (one or more threading models, in- and out-of-process execution and distributed execution) [Rogerson97]. Following is a brief description of COM’s concurrency support, as described in Inside COM [Rogerson97] and Inside Distributed COM [Eddon98]. The aim of this description is to explain in very brief terms the fundamental problems of concurrent programming. Two fundamental problems have to be solved in concurrent programming. The first one is to ensure safe execution of critical code, or to synchronize different threads of execution to ensure that resources remain in a consistent state. The second problem is how to pass data across address space boundaries, which can be a non-trivial task, especially when address space boundaries lie between two incompatible hardware or operating systems.

COM takes the all-or-nothing approach to synchronization; it handles it completely for you or not at all, in which case synchronization primitives provided by the operating system have to be used. The former approach works to the extent that COM components that are completely unaware of any threading model can still be executed in a multithreaded application.
This is the approach COM people call apartment threading. There are two kinds of apartments; single-threaded apartments (STA) and multithreaded apartments (MTA). In single threaded apartments, you get the “all” approach of COM. Components executing in a STA can be completely thread oblivious, as COM ensures that all calls into such an apartment are synchronized. Calls inside a STA are synchronized by definition, as there is only one thread of execution. COM achieves this by employing window message queues.

All external calls into a STA are done automatically through proxies that put the calls in a single queue, which are in turn removed by the STA message loop, and executed sequentially. Figure 9, which is adapted from Figure 4-2 in “Inside Distributed COM” [Eddon98], shows the communication path from an external client to an object that resides in a STA. Thick arrows denote thread boundaries.

Although calls to the STA are synchronized in this way, the story does not end here, as method calls generally have parameters that need to be passed, possibly across address space boundaries. Such parameters can contain component references that are accessible outside the target apartment (thread), which means that some steps have to be taken to ensure safe operation involving such parameters. COM solves this by employing local-RPC (shown in Figure 9), which means that parameters in cross-apartment calls are marshaled. See [Eddon98][Rogerson97], [Tanenbaum92], or any literature on RPC for further details on marshaling.
The MTA model is the "nothing" part of COM’s threading support. It is best described as a free playground for components using threads, where each component is responsible for its own safety, using synchronization primitives provided by the operating system.

Notice that from the perspective of components as presented here, there is no logical difference between inter-process calls, or cross-computer (distributed) calls. The only difference is that distributed calls take a longer time to complete.

**Motivation**

Concurrency touches the subject of event models in two places, at least. First, an event model is one of the means by which concurrent threads of execution can communicate. A prime example of this is COM’s apartment threading model (see Description). If intended for cross-thread communications, the event model itself has to support concurrency, that is, be thread safe. Another issue is that of cross-process (out-of-process) event delivery in the case of events carrying parameters. In that scenario, the event delivery mechanism

---

**Figure 9, Inner workings of the STA model**

The inner workings of the STA model
has to support parameter marshaling for delivering parameters to a different address space, or rely on an underlying mechanism, such as RPC, that does.

Second, the event model itself may employ concurrency techniques for providing advanced event services such as composite event detection, event logging, event filtering, etc. An example is complex event detection, for which several concurrent schemes have been suggested. [Jaeger97][Liao98].

The level of concurrency provided is important when considering issues like maximization of responsiveness and utilization of resources (like multiple processors). On the other hand, high levels of concurrency might have a negative impact on other factors, like simplicity of use and effort needed for system integration.

Additionally, concurrency is known to complicate matters in event models, which is not surprising. One issue is well known in particular, and that is multicasting in the presence of threading and re-entrance [Szyperski98]. In the absence of countermeasures, the result of an event multicast is unknown in the case of someone subscribing/unsubscribing to an event while a multicast is being performed. See also consistent multicast in the next chapter.

Criteria

Consistent multicast

Concurrent detection/signaling

Simultaneous event occurrences
4.2.5 Semantics

1: the study of meanings: a: the historical and psychological study and the classification of changes in the signification of words or forms viewed as factors in linguistic development b: SEMIOTIC (2): a branch of semiotic dealing with the relations between signs and what they refer to and including theories of denotation, extension, naming, and truth
2: GENERAL SEMANTICS
3 a: the meaning or relationship of meanings of a sign or set of signs; especially: connotative meaning b: the language used (as in advertising or political propaganda) to achieve a desired effect on an audience especially through the use of words with novel or dual meanings

[MW]

Description

The semantics of a component defines the actual meaning of running the operations of a component, whereas the syntax of a component defines how the operations of a component are invoked. The semantics of a component are frequently specified in the contracts of interfaces that the component implements (see chapter 4.2.1). A semantic problem arises when multiple equally valid semantics exist for a given functionality, for example when both breadth-first and depth-first policies can be used, but resulting in somewhat different semantics.

Motivation

All current component wiring standards provide some method for asynchronous notification [Szyperski98]. A mechanism that provides such functionality has to address some subtle issues.
Preservation of causality is an issue when notifications can have multiple recipients, and recipients are free to forward received notifications [Szyperski98]. In such a scenario, race conditions can easily arise. In the example illustrated in Figure 10, race conditions will arise when C1 signals its OutgoingEvent (assuming that C2 will forward the signal to its own OutgoingEvent). If the results of C3 differ depending on who the source of its incoming event is, the results will depend on the model used to propagate the event. If the event is propagated in a breadth-first order, which preserves causality, C3 will get its notification from C1. On the other hand, if events are propagated in a depth-first order, the actual event source for C3’s notification will depend on whether C2 or C3 has a higher priority in event delivery, which would be determined by the order in which they subscribed in a naïve implementation.

Another problem when using asynchronous notifications is the inconsistent state of a system while a multicast is being performed. When one component has received a notification that changes its state, it is reflecting the state of the system after the event that triggered the notification occurred. At the same time, another component that has

![Figure 10: Problem of asynchrony](image-url)
not received the notification is reflecting the system as it was before the event occurrence. This can be observed by components using regular method invocations to examine other components [Szyperski98]. Errors that are caused by this problem can be very hard to track down, especially when event subscriptions are dynamic and used by third parties in combinations that cannot be foreseen by the component developer. It is not clear to the author whether this is a problem that can or should be tackled by the event model, but it is included here to point it out as a potential problem in event models.

Third, it is possible that while a multicast is being performed, the set of recipients is changed by subscription/unsubscription requests, unless special measures are taken to prevent such actions. In this dissertation, the term consistent multicast will be used for multicast specification with such measures in place.

Fourth, errors and exceptions raised by notification recipients effect the semantics of event delivery. Should the delivery algorithm ignore the exception and continue, or propagate the exception to the originator of the notification? Either way, difficult design decisions have to made, for instance, how should the exception be propagated to the originator if event delivery is being performed in a separate thread from event detection? The issues of exception/error handling by the event model also arise under the Safety factor and are covered in more depth there.

Although the issues above contribute towards the notion of atomic event delivery, fault tolerance and consistent multicast are not enough to ensure atomicity. In addition to those, a protocol that allows the event model to cancel event delivery if it fails to deliver to any of the recipients. This bears considerable resemblance to database transactions, which would suggest a transaction-like protocol for ensuring atomicity of event delivery.
Criteria

- Preservation of causality
- Consistent system state
- Consistent multicast
- Fault tolerance
- Atomic event delivery

4.2.6 Safety

1: the condition of being safe from undergoing or causing hurt, injury, or loss
2: a device (as on a weapon or a machine) designed to prevent inadvertent or hazardous operation

Description

The property of a system to prevent certain failures either by statically eliminating classes of errors or by dynamically detecting errors and handling them to avoid failures.\[^5\]

[Szyperski98]

One of the fundamental properties of components is that they are subject to third party construction and composition [Szyperski98]. It follows that the safety of a system is not defined by a single party, as each party involved in constructing a component defines the safety of the component. If a system composed out of such independently built components is to be safe, some assertions about the components, or the way by which they are composed, have to be possible.

\[^5\] Note that here is a certain discrepancy between the usage of the term safety here, and in discussions on safety critical systems. In safety critical systems, the term refers to the safety of external factors such as property and life (as in avionics systems) and the effect of a software system thereon, while here it refers to the safety of the software system itself, and the effect of a component thereon.
In the absence of countermeasures, a component system is only as strong as its weakest component. Fault isolation is thus an essential theme, as is safety of individual components.

[Szyperski98]

In general, safety can be ensured in two ways. One is to prevent failures from occurring by taking precautionary steps. This requires a certain degree of foresight, as only pre-known classes of faults can be thwarted by precautions. Certain fault classes can be eliminated by static checking, which has to be supported by the utilized programming language [Szyperski98]. Error classes that can be eliminated in this way include memory errors, type errors (type safety) and access errors (module safety) [Szyperski98]

The other way is to take corrective measures after a fault has occurred to prevent it from escalating to a failure. First, a fault has to be detected before corrective steps can be taken. Not all fault types can be detected however, and not all faults that can be detected

Figure 11 Classification of Failure Models (adapted from [Hadzilacos93] )
can be corrected. For a classification of fault types, see [Hadzilacos93][Schneider93].

When a fault has been detected, it is either handled locally or propagated to the caller as an error. The traditional way of propagating errors is by using error codes in return values, such as COM’s HRESULT codes. Errors are also frequently propagated using exception handling. Problems arise when the component that the error should be propagated back to, called the service in an asynchronous way. In that case, no direct return path exists (i.e. the call stack), and the error has to be signaled explicitly in a similar way to the calling of a service.

**Motivation**

Certain methods used for event delivery are known to defeat static safety checking performed by compilers [Szyperski98][JavaBeans]. An example of an event delivery mechanism that defeats static type checking is a generic event demultiplexer described in [JavaBeans]. It is thus important to be aware of such shortcomings in event models and circumvent them if possible. Also, where static type checking is not available, its run-time counterpart has to be available for ensuring safe operation of the event model.

**Fault tolerance.** An event model is one of the ways components can use to communicate. As such, the event model is on the border between components, which means that errors crossing component boundaries will go through the event model in one or another form. The event model will therefore have to be able to cope with component failures and support the handling of errors. The event model can take on three important roles in this context:

- Error propagation.
- Error isolation
- Error logging
Two of these, error propagation and error isolation are opposites, either but not both of them is in place for a given event subscriber. It might be an option which one is chosen if both are supported. Error logging can be performed irrespective of whether failures are isolated or propagated as errors. The main issue here is that it should be well defined how failures that occur during event model operations are handled.

As for the issue of module safety/security, it is deemed as a too wide subject to be covered adequately here, and is thus left for future work. Areas that need to be covered here are, amongst others, language support for static module checking in terms of events, run-time checking of event invocations, and authorization mechanisms.

**Criteria**

- **Static type checking**
- **Run time type checking**
- **Exception handling**

**4.2.7 Service level**

**Description**

The service level of a component model refers to the quality of service delivered by various parts of the component model. This is also known as non-functional requirements, such as availability, mean time between failures, throughput, latency, capacity and so on [Szyperski98]. These are requirements that do not relate directly to what should be done, but rather how well it should be done. This dissertation will not attempt to cover all known service level issues, but will instead focus on two that are amongst the most important; namely efficiency and accuracy. Others could also be
included, throughput and latency are for example highly relevant, but perhaps better suited to a discussion of network protocols, which is not the focus of this dissertation.

Software efficiency is often measured with execution complexity, the cost of execution as a function of the input. This can be anything from a constant (cost does not depend on the input) to exponential growth in cost. In distributed settings, other methods of measurement are sometimes more useful, such as round-trips needed to perform an algorithm [Rogerson97][Box98], as network round-trips are typically several orders of magnitude more expensive than local operations.

The time at which the cost is incurred is also of significance. The steps needed to eventually perform an operation at the request of a user are several, frequently referred to as compile-time, link-time, load-time and run-time. Those steps are, respectively, the time at which the source code is compiled into binary form, independent modules are linked together, the application or independent modules are loaded into memory, and finally the actual execution of the application. It is important to observe that linking can occur after other parts of an application have already been loaded, which is the case when dynamic link libraries are used, as this blurs the line between load-time and link-time.

Dynamic linking is one of the corner stones of component technology [Szyperski98][Rogerson97]. In general, it is desirable to have the cost of an operation to be realized as soon as possible. For instance, almost every compiler today has the option to optimize its output at the cost of increased compile time, but gaining improved efficiency at load- and run-time. However, the opposite may also be true in some cases. For instance, the initialization of an operation that is expensive to initialize (e.g. in a distributed setting) might be deferred until the operation is required, to decrease the
load-time of a module and improving overall performance. This practice is known as very late binding [Szyperski98]. Very late binding has several implications for application development, integration testing may for instance become impossible when this approach is employed [Szyperski98].

Accuracy denotes how relevant the results of a computation are, no matter the nature. The term is naturally best understood in the context of mathematical computations. It is for example very common for floating point operations to perform approximations and in many cases those approximations become more accurate with increased computation time. Accuracy also applies to other kinds of algorithms. An example from the database world is the two-phase commit algorithm that is meant to ensure atomic commitment over a distributed database. Researchers found out that two-phase commit did not guarantee what it was meant to do, i.e. it wasn’t accurate enough, and thus developed three phased commit which is more accurate.

**Motivation**

The event model defines the method used to deliver events, and provides the infrastructure used for delivery in some cases. In such cases, it also dictates the service level provided by event delivery mechanisms in a given component model, and is thereby a factor in determining the applicability of the component model in a given scenario.

On one hand there is the efficiency of event processing: the consumption of computing resources, including processing power and network bandwidth, necessary to deliver an event from source to sink. In a local setting, this is mainly a question of execution complexity. In a distributed setting, other factors affect performance more, such as network round-trips required for setup and execution.
On the other hand, there is accuracy. In the case of events, accuracy is the question of guaranteed event delivery. In most cases this is a question of underlying protocols, such as RPC, but an event delivery mechanism might also define its own delivery protocol, possibly using a mixture of low level protocols [Liao97], in which case delivery guarantees are defined by the delivery protocol. Accuracy might need to be sacrificed for overall system performance, for example, using upper bounds on the time allowed to deliver an event would improve performance but decrease accuracy.

Other non-functional requirements than those mentioned above may also be relevant, but will not be covered further here.

**Criteria**

**Performance**

**Accuracy**

4.2.8 Heterogeneity

*Main Entry:* heterogeneous  
*Pronunciation:* "he-tə-rə-"jE-nə-səs, "he-trə-, -nyəs  
*Function:* adjective  
*Etymology:* Medieval Latin heterogeneus, from Greek heterogenEs, from heter- + genos kind -- more at KIN  
*Date: 1630*

: consisting of dissimilar or diverse ingredients or constituents : mixed [MW]

**Description**

Heterogeneity is the quality or state of being heterogeneous [MW]. Heterogeneous systems have to deal with several problems that follow the use of dissimilar systems that are more or less incompatible. Heterogeneity in software systems emerges on several
levels. In this paper, three “levels” of heterogeneity are defined, although more can certainly be found.

The first level is the platform level. Platform level is the lowest level of abstraction that a developer may access, such as an OS / hardware combination, or a virtual machine like the Java VM. In computer science, this is what the term “heterogeneous” refers to in most cases. Platform heterogeneity presents many challenging problems that have received a lot of attention in the literature. Typical of such problems are those that have to be addressed by RPC, like endian-ness and numerical representations [Tanenbaum92].

The second level is the component model level. A large software system may have to operate on the terms of more than one component model, i.e. both COM and CORBA, thus introducing heterogeneity at the component level. Bridges between component models are one way of solving this problem.

The third level is the language level. Large software systems are frequently implemented using a mixture of languages, thus introducing heterogeneity at the language level. This introduces several problems, like the need for standardized data type representation at the binary level and standard call conventions.

**Motivation**

Event models have been identified as a mechanism that can be used to “…support push/pull propagation of data in a distributed environment.” [Le98]

As one of the primary issues in many distributed environments is the heterogeneity of the participating platforms, the ability of an event model to support such heterogeneous platforms is a deciding factor in fulfilling this potential. One of the ways that this can be accomplished is to build the event model on an established infrastructure enabling communication between heterogeneous platforms, such as DCE RPC, or CORBA. As
this suggests, the ability of an event model to address heterogeneity is not necessarily dependent on the design of the event model itself, but rather the underlying technology used to deliver events.

Event sources themselves are an important source of heterogeneity, and actually one of the motivations for this work. Different event sources are generally not compatible because of different interfaces and/or method signatures, even within the same event model. Additionally, event models of different component models are generally not compatible. Ideally, an event model would provide a consistent view and use of events across component models. An alternative is to build bridges between event models of different component models, to present a coherent view of all accessible events.

Heterogeneous languages. As events in component models are expressed in terms of the component model (e.g. in IDL), events are usable in all languages supporting the component model in question. The question of which languages a given event model can be used with, is thus the question of language support for the component model in question.

Criteria

Heterogeneous platforms

Heterogeneous component models

Heterogeneous programming languages
5. Evaluation of the CORBA Event Service and the COM IConnectionPoint event model

5.1 Introduction

This is an evaluation of the CORBA and COM specifications, with respect to their suitability to support a Snoop-style complex event detection service. The basis for an event service in CORBA is the one defined in the CORBA Services document [CORBAser], while the basis in COM is the connectable objects mechanism.

It should be clear that an evaluation such as this can never be complete. First of all the evaluation is performed on the specification level, which means that certain factors, especially non-functional ones, cannot be evaluated without making assumptions about implementation choices. Evaluation of such factors is therefore kept to a minimum. Second, to be able to evaluate the suitability for supporting complex event detection, certain design decisions have to be made. This will be made clear wherever possible, especially where alternative ways have been identified.

This evaluation is based on requirements identified in the definition of Snoop [Cha93]. Where additional requirements described in related literature are known to the author, they are also included, but completeness in that respect is not claimed.

Finally, as the evaluation is made on the specification level, it aims at identifying to which extent the specifications address individual requirements presented in this paper. In some cases it is however not possible, due to the nature of the requirement or because of time constraints, to assert whether the specification is sufficient, i.e. whether it defines enough functionality to address the requirement in question. An example of this is module safety, which cannot be addressed here in detail due to time constraints.
Note: Where the term “designer” is used in subsequent chapters, it is referring to the designer of the complex event service, unless context makes some other use of the term obvious.

5.2 Evaluation

5.2.1 Contracts

Requirements

Following is what the author has identified as contract requirements that a complex event service puts on the component and event models, along with general requirements identified in the framework.

Event delivery semantics. Events are in many cases delivered with a multicast protocol. The semantics of that protocol must be well defined. This issue is further covered in the chapters on concurrency, semantics and service level.

Event parameters. Events have standard and user defined parameters. The form & semantics of those parameters have to be well specified. Snoop specifies three distinct elements that have to be included in an event contract. Two of them are mandatory; an event has to carry with it, or make otherwise identifiable, the event type of which it is an instance, and a timestamp that denotes the time of its occurrence. The third element, the parameters of the event, is optional. In a relational database environment, such parameters would typically consist of related tuples and the operation triggering an event. An additional element that has been suggested as necessary in a database context is the transaction identifier of the transaction from which the event originates [Eriksson98]. Snoop defines four distinct consumption modes. Some of them, like the cumulative consumption mode, can result in considerable storage requirements, thus
necessitating that event instances be persisted. Given that the event service supports
event instance persistence, restrictions (if any) on event parameters resulting from the
persistence support need to be specified. For instance, it may be inadvisable to allow
large bitmaps to be passed as event parameters when event instances are being persisted.

Event interfaces. Events need to be subscribed to, by other events in the case of
complex events and by rules that are triggered by events. Thus, interfaces for
subscription/unsubscription need to be specified.

Event versions. In the context of components and their late integration, as is the
context of event notifications in this dissertation, versioning of event interfaces is a
requirement put on the event model irrespective of the application.

Snoop is an event specification language, so it also requires mechanisms for
interpreting/compiling event specifications and building event detection trees. However,
this dissertation focuses on the event service that is needed to support the specifications
expressible in Snoop, not the language itself. The contract requirements of
interpreting/compiling Snoop are thus out of scope for this work.

**COM support**

Event delivery semantics. The COM specifications do not specify the semantics of
event delivery. This topic is covered further in the chapters on concurrency, semantics
and service level.

Event parameters. As illustrated in the background on the COM connection points
model, an event interface is specified by the application programmer from the ground up,
requiring only support for the most basic interface, IUnknown. Thus, all parameters
associated with an event have to be defined from scratch. Then there is only the question
of data types available for specifying event interfaces. As put forth in the requirements
above, we have to be able to specify three types of attributes: a timestamp, the event type and user defined parameters.

As COM defines a date/time datatype as a part of the Variant data type, it seems logical to use this standard date/time datatype for the timestamp parameter.

The second attribute is the event type. The only requirement is that it has to have a sufficiently large domain to be able to assign a unique value to every event type. Obvious candidates are integer and string types.

The third attribute is a little more problematic. The nature of user definable parameters is, of course, that they are not defined by the event model, but have to be supported nonetheless. COM defines the Variant data type, which can be used for this purpose, as it can hold an array of values, which can be user defined. A variant can also hold an object reference to any COM object, which again can be user defined. This can however present new problems, e.g. if the event history is to be persistent.

An example of an interface definition for supporting Snoop style event notifications can be found in Appendix B.

Event interfaces. COM connection points define an elaborate design around event subscription/unsubscription, which can be used in a complex event service. Thus, little effort on this part is required.

Event versions. Versioning is one of the cornerstones of COM. The only requirement put on an event interface in COM is that it is exactly that, an interface. This means automatically that it has to have an interface identifier, which serves also as the version checking mechanism in COM. Thus, versioning of event contracts is implicitly supported in COM.
CORBA support

Event delivery semantics

This topic is covered in the chapter on concurrency, semantics and service level.

Event parameters

Unlike COM, the CORBA event channel defines how event parameters can be passed, with the definition of the generic event channel. The generic event channel specifies that data that can be represented with the Any data type can be passed along as event parameters. This includes data types that can be used to represent parameters required by Snoop, namely event type and timestamp, along with user defined parameters. The event contract must specify how those parameters are packaged into the Any parameter, both to allow event suppliers to pass the parameters correctly, and also so the event service knows how to retrieve them. As CORBA does not state in its specifications how date/time values should be represented (various options are given), the format of the timestamp also has to be included in the event contract.

Subscription/unsubscription.

By using event channels, the specification of how subscription/unsubscription is performed has been obtained from the CORBA services specification, and can be used unchanged.

Versioning

CORBA supports three different versioning schemes through interface repository RepositoryIDs. Those are DCE UUID’s, simple incremental version numbers and named versions. All of those can additionally include minor versions denoting backwards compatibility. Typed event channels define event notifications through a specially defined interface, which can thus have its own version in the interface repository. The generic
event channel does however define its own interfaces for event notifications (i.e. 
PushConsumer/PushSupplier), which the user does not have control over, and can thus 
not support versioning in this manner.

**Summary**

Neither COM connection points nor CORBA event channels define atomic delivery of 
events. Using connection points places no constraints whatsoever on the contracts 
defined for such a protocol, while using event channels, the contracts would have to be 
compatible with the existing event delivery specification.

Connection points do not define event delivery in any way other than requiring that 
notifications are defined by a interface, leaving the specification entirely to the user. 
COM offers datatypes that can be used to convey all the parameters required by Snoop. 
Generic event channels define event delivery with the Any data type, leaving details of 
packaging parameters to the designer. CORBA lacks a standard data type for date/time 
values, leaving the definition of the timestamp parameter to the designer.

Both event channels and connection points define adequate mechanisms for 
subscription/unsubscription.

Both COM and CORBA support versioning of event interfaces. The use of the generic 
event channel in CORBA defeats versioning.

5.2.2 Polymorphism

**Requirements**

Polymorphic event notifications. Composite event detection in Snoop is performed by 
building event graphs, or event trees, where each node in the tree is a subscriber to 
events occurring in nodes lower in the tree, and signaling its own event when a
A composite event is detected. Given that composite events can be composed of events with different parameters, this indicates that event notifications or their parameters have to be polymorphic in some way.

**COM support**

**Polymorphic event notifications.** Event interfaces are not defined in COM. Each event is frequently designed with its own interface from scratch, which leads to non-polymorphic events in the general case. As traditional methods are not polymorphic, some form of parameter packaging is needed for providing parameter polymorphism. This can be performed in a proprietary way, or by employing existing approaches. COM defines the IDispatch interface that can be used for this purpose. IDispatch defines how parameters are packaged and then sent through one method, the *Invoke* method. As dispatch interfaces support run-time discovery of parameters, i.e. the receiving Invoke method does not have to know beforehand the parameters it receives, a complex event service can discover them at runtime and store them in its own format for later use, or persist them if that is required.

**CORBA support**

**Polymorphic event notifications.** CORBA generic event channels define that event data should be passed using the Any datatype. As the Any datatype supports packaging of other data into it, i.e. Any can hold a complex value, generic event channels are polymorphic. Generic event channels are not the only type of event channels allowed in CORBA, the event service also defines typed event channels. Typed event channels can be handled in a general way by employing the Dynamic Skeleton Interface (DSI), which allows an object server to handle all object requests through a single method.
implementation using parameter packaging.

**Summary**

Events in both CORBA and COM are polymorphic to some extent, although on different basis. COM events are polymorphic if they use dispatch interfaces, CORBA events are inherently polymorphic if they use the generic event channel. Additionally, typed event channels in CORBA can also be handled polymorphically using DSI. If supporting existing event sources is not an issue, polymorphic events are easily achievable in both systems.

**5.2.3 Encapsulation**

**Requirements**

Event source encapsulation. Snoop specification requires separation between physical and logical events, where there is a unique mapping between logical and physical events. One way of viewing this is to look at logical events as encapsulating physical events.

This requires event chaining, where the occurrence of one event will lead to the occurrence of another, thereby making it possible to hide the former event by exposing only the latter one.

Event parameter encapsulation. As described in motivation, protection or encapsulation of event parameters is necessary in event multicast.

**COM support**

Event source encapsulation. COM does not support this capability specifically. It is possible to implement however, but requires a special encapsulating class in the general case. By imposing some additional restrictions on the use of the event model, it is
possible to support event encapsulation in a general way (see more on this in the
discussion on polymorphism).

**Event parameter encapsulation.** COM places no restrictions on the parameters of event
notifications, thus giving the user the option of using a scheme that can lead to
inconsistent delivery of events. This includes the use of in/out parameters and passing of
objects that does not encapsulate its data adequately.

### CORBA support

**Event source encapsulation.** The CORBA Event Service Specification addresses event
chaining by employing event channel objects. The specification defines two types of
event communication: typed and untyped. In the case of untyped communication, where
there is only a single parameter of type `any`, the generic event channel can always be
employed. When handling typed events, we get into a similar situation as with COM
`IConnectionPoint`; a custom made event channel needs to be built to encapsulate such
events.

**Event parameter encapsulation.** CORBA requires parameters of event notifications to
be unidirectional and they are always passed by value. CORBA objects cannot be passed
by value, thus restricting event parameters to standard datatypes. This obliviates the need
for encapsulating event parameters, as event recipients cannot effect the parameters
other recipients receive.

### Summary

The CORBA event service specifies an event encapsulation mechanism in the form of
Event Channels, while COM specifies no comparable mechanism. It is possible to build a
general mechanism for COM for a subset of events (those using the `IDispatch` interface),
although it is not straightforward. In other cases, special classes would be implemented to support encapsulation of individual event suppliers.

CORBA obliterates the need for encapsulating event parameters, while COM leaves open options that may endanger consistent reception of event parameters by all recipients.

5.2.4 Concurrency

Requirements

Snoop has several requirements when it comes to concurrency. For one, as has been stated before, complex event detection requires support for multicasting from the event model. As multicasting is not by default a thread-safe operation, as stated in the motivation in 4.2.4, consistent multicasting has to be explicitly supported, either by the event model itself, or by participating components that act as event sources.

Although arguably not a requirement, Snoop supports the notion of performing event detection and event signaling in separate threads. Additionally, concurrent schemes for complex event detection have been suggested in the literature [Jaeger97]. To be able to support these schemes, the event model and its environment (the component model) have to support concurrency.

Simultaneous event occurrences. In distributed settings such as supported by the event models in question here, simultaneous occurrences of events, i.e. events whose relative ordering cannot be established, cannot be ruled out. Although handling of simultaneous events is not defined by Snoop, such extensions are suggested and must be considered in concurrent settings. If a complex event detector is to be capable of identifying and handling simultaneous event occurrences, a “universal” time service has to be supported,
which is able to supply the system wide time along with an error estimate.

**COM support**

**Consistent multicast.** As COM does not specify how the actual notification of events is performed (although it does define the IEnumConnections interface whose purpose is to support multicast), it clearly does not define how consistent multicasting would be performed. It is therefore the responsibility of each event-emitting component to define a thread-safe event notification scheme.

**Concurrent detection/signaling.** As can be seen in the introduction on concurrency in chapter 4.2.4, COM has extensive support for concurrency. DCOM defines extensions for supporting distributed concurrency models, and COM itself defines mechanisms for both in- and out of process concurrent invocation of components. COM does not, however, provide synchronization primitives required for free-threading components, but relies on the operating system for such support. As the Win32 API supports all synchronization primitives required in the general case (including semaphores and thread events), concurrent event detection schemes should be deemed feasible in COM.

**Simultaneous event occurrences.** As described above, support for simultaneously occurring events in a complex event detector requires global time. Neither COM, nor its distribution extension, DCOM, specifies such a service. Thus it has to be concluded that such a service would have to be acquired from a third party or defined specifically for this purpose.

**CORBA support**

**Consistent multicast.** Although support for distribution, and thus concurrency, is one of the design goals stated for event channels in CORBA, a policy supporting consistent
multicast by event channels is not described. Thus, such a policy would have to be
designed for a complex event detector utilizing event channels.

**Concurrent detection/signaling.** One of the CORBA services [CORBAser] is the
concurrency service. The concurrency service provides concurrency control for arbitrary
resources, it is the responsibility of the client to identify resources that need concurrency
control and protect them using the service. Moreover, the service is intended to work
also in transactional context. Regarding the general nature of this service, the support
provided is likely sufficient for concurrent detection/signaling, but that will not be
determined here.

**Simultaneous event occurrences.** Another of the services specified in [CORBAser] is
the time service. It specifies basic time services such as acquisition of global time, and
comparison of time values given a time measurement inaccuracy. Those are the services
needed for supporting detection of simultaneous events.

**Summary**

Neither COM nor CORBA define consistent multicasting explicitly. Both provide
extensive support for concurrency control, both local and distributed. COM does not
specify any global time services, while CORBA does.

5.2.5 **Semantics**

**Requirements**

Complex event detection places the following semantic requirements on the event
model, semantic requirements of complex event detection itself are not included. Also,
those are semantic issues that have to be considered irrespective of the application.
Event delivery order (preservation of causality). In [Cha93], event delivery order is not explicitly specified, and the issue is not mentioned. However, the algorithm listed in the paper uses depth-first ordering. One should, however, interpret that openly, as the algorithm given is an example used to clarify the concept of complex event detection, and depth-first ordering is significantly simpler to implement and understand than breadth-first ordering. The view taken here is that complex event detection does not make a requirement of either ordering.

System consistency while delivery is in progress. One of the semantic problems that may arise in event based applications, is ensuring system consistency while event delivery takes place. As mentioned in the motivation (4.2.5), it is not clear whether/how this problem should be tackled by the event model, and thus is left out of this evaluation for future work.

Delivery consistency (consistent multicast). As described under motivation in 4.2.5, consistent multicast is a multicast where the behavior of the system is well defined under arbitrary subscriptions/unsubscriptions while notifications are being performed. A common technique (employed by the Java event model for one) is to make subscription lists immutable while event notifications are being performed. This is a requirement on the event model irrespective of the application, if dynamic subscriptions/unsubscriptions are allowed (which is mandatory in a component system) the event model has to address this issue.

Error/exception handling.

Yet another issue which affects the semantics of multicasting is the occurrence of errors/exceptions when a multicast is being performed. The behavior under such
circumstances has to be well defined. This issue is covered further in the chapter on Safety.

**Atomic event delivery.**

A general problem when using event models that support multicast, is the one of atomic event delivery. This is a problem particularly relevant to Snoop-style complex event detection, given the definition of an event (**Definition 2**) put forth in [Cha93]. From a observer’s point of view, the atomic property of an event is not fulfilled unless multicast is guaranteed to be atomic.

**COM support**

COM does not specify how notifications are performed, and thus does not address any of the issues specified above in requirements. The requirements above are therefore not resolvable on the specification level of COM.

**CORBA support**

The CORBA event service defines a generic event channel, which includes event multicast. However, the specification only dictates the syntax of event notifications, i.e. semantic issues such as those listed above under requirements are left from the specification. It is thus dependent on the implementer of the event service how those issues are resolved, and thus falls outside what can be evaluated on the specification level.

**Summary**

Neither COM nor CORBA specifications can be evaluated in terms of semantics, as neither of them covers semantic issues.
5.2.6 Safety

Requirements

Safety is not a requirement put forth in the definition of Snoop. However, it is a requirement that can be put on event services in general as described under motivation (4.2.6). The question is mainly which, if any, constraints the service puts on safety features such as static type checking, safe memory handling and exception handling. No requirements, other than those identified in the motivation section, have been identified for Snoop-style complex event detection by the author.

COM support

Preventive.

Static type checking. COM itself does not provide static type checking, it is a feature of the programming language being used. Type checking for events has to be done in two steps: First is the question of checking the safety of event subscriptions. Second is the question of how event signaling, which is usually in the form of method calls, is type-checked.

Events subscriptions in COM are not statically type checked, as the subscription is always performed at runtime, and the nature of the IConnectionPoint interface which handles subscription requests rules out static type checking of event subscriptions in languages such as C++. Whether static type checking is provided in other languages, or whether it is possible in the first place, is out of scope for this work. Event signaling is usually statically checked, as the interface used for signaling is known at compile time.

Run time type checking. COM does provide strong run-time checking facilities, built around its interface paradigm. Events can be checked with this mechanism; it is the party
who implements the event emitting component that decides whether this is done. COM thus supports, and recommends this type of type checking.

**Corrective.**

**Exception handling.** It is up to the source component to determine how to respond to errors from its sinks, which is the complex event detector in this case. COM requires that operations in COM interfaces always return the so-called HRESULT codes, which indicate success or failure. COM does not specify any other means for handling exceptions. An event source in COM thus has to define how it responds to error codes from its sinks. Options include to ignore them, to halt notifications on an error and report it to the connection point, or to continue notifications, in which case it is necessary to collect all errors by some means and make them available to the sender. As COM leaves the actual event notification, and its specification, up to the user, any of these policies can be adopted.

**CORBA support**

**Preventive**

**Static type checking.** As CORBA is not a language-specific standard, it depends on language mappings for static type checking. As stated before, type checking of events happens in two steps: checking of event subscriptions and checking of event signaling. Static type checking of event subscriptions is not possible due to the model used for subscription/unsubscription, i.e. the interfaces used for those operations are not type specific. Static checking of event signaling is however possible in some cases, i.e. when using typed event channels. In the case of generic event channels, the event signaling method is standardized, taking only one parameter of type Any, which cannot be statically type checked (static checking would defeat the purpose of such a datatype).
Run-time type checking. CORBA provides run-time type checking facilities based on RepositoryIDs, which are also known as type identifiers. RepositoryIDs are analogous to interface identifiers in COM. These facilities are used where static type checking is not possible, e.g. in the case of subscriptions in typed event channels.

Corrective

CORBA supports the notion of exception handling rather than the return of error codes. Interestingly, the CORBA event service does not specify how event channels should react to exceptions raised by event consumers in the push model. Thus, it is probably assumed that an exception raised in a consumer halts event delivery and returns control to the closest exception handler. The options described in the chapter on COM support are also available to the designer of a custom event channel.

Summary

Both COM and CORBA event models have similar limitations in the case of type checking of events. Event notifications can be statically type checked, except for the generic event channel, while event subscriptions cannot. Both offer similar mechanisms for run-time type checking to help ensure safe event subscriptions. COM error handling is based on function return values (HRESULT) while CORBA uses exception handling. Neither specifies event notification semantics in the presence of errors/exceptions.

5.2.7 Service level

Requirements

In [Cha93], two service level requirement are mentioned: timeliness and atomicity. Although no absolute requirements are given, timeliness put forth the requirement that detection of events should be detected effectively, i.e. should not require great amounts
of processing power, or in the case it does, should be provided sufficient processing power to fulfill timeliness requirements.

The requirement for atomicity is a direct result of the definition of an event used in [Cha93]: “An event is an atomic (happens completely or not at all) occurrence”. In the presence of multicasting, this has an interesting consequence. If this requirement is interpreted as guaranteed atomic delivery, it has the consequence that event delivery has to be implemented using a protocol that can ensure such a service level.

**COM support**

**Performance.** On the issue of timeliness, that is clearly an implementation issue. As this is an evaluation on the specification level, this cannot be asserted there.

**Accuracy.** On the issue of atomicity, this is more of a functional requirement, and could be included in specifications. However, as stated before, COM connection points do not specify how event delivery is performed, and thus in particular does not specify atomic event delivery. On the other hand, there is nothing in the specification that would prevent atomic delivery of events.

**CORBA support**

**Performance.** Nothing can be said about timeliness of event services based on the CORBA event channel specification, as timeliness is clearly a function of an implementation. However, it should be noted that in [Le98], it is stated that “This [the use of multiple event channels] is likely to have an effect on the performance, especially because the Event Channel has more than one sub-component that needs to be created for each instance of an Event Channel.”. In other words, it is suggested that the design of event channels may prevent the construction of effective complex event detection around
the event channel concept.

Accuracy. Atomic event delivery is explicitly not supported by CORBA event channels. Any design offering guaranteed event delivery would have to define its own protocol for this purpose. Although the author is not aware of any protocols designed exclusively for atomic event delivery, such a protocol would probably be something along the lines of two-phased commit. As the CORBA event service specification already defines non-atomic event delivery, the protocol would have to be compatible with that definition. Although service level factors are not included in the CORBA event service specification, it is designed with various service level factors in mind, such as fail-over safety, security and others [CORBAser].

Summary

Neither CORBA nor COM specify the level of service provided by their event services. The CORBA event channel is designed with various service factors in mind (without specifying them explicitly) including fail-over safety, security, and others.

5.2.8 Heterogeneity

Requirements

[Cha93] does not define any requirements for heterogeneity, such definitions are out of the scope of that paper. The need for heterogeneity support arises from the need for mixing of platforms/component standards/programming languages into a single system, both to integrate existing systems, and to develop new systems that leverage the strengths of different platforms. As this is a pretty broad requirement, the best that can be achieved here is to enumerate supported platforms/languages. Heterogeneous component models are excluded here, as this is an evaluation of the event models in two
component models, which are obviously not aimed at heterogeneous component models. Although the evaluations presented here are mostly based on specifications, this chapter presents a brief overview of existing implementations to give an idea of heterogeneity issues.

**COM support**

**Platforms**

COM is a technology designed and implemented by Microsoft Corporation. Initially, and until recently, it was only available on the company’s own platforms, the Windows family. However, because of Microsoft’s overwhelming market share in operating systems and office productivity software, COM has become a de-facto standard, and other software companies have made moves to support the technology on other platforms such as UNIX and mainframes.

SoftwareAG’s EntireX DCOM is a product enabling COM based development across platforms. Currently supported platforms are OS/390, Sun Solaris, TRU64 Unix, IBM AIX, HP-UX and Linux [http://www.softwareag.com/entirex].

Bristol Technology’s Wind/U® “is a set of libraries and utilities that enable you to create native UNIX, OpenVMS, and OS/390 applications from your Microsoft Windows API and Visual C++ source code” [http://www.bristol.com]. It includes support for COM and ActiveX technologies, but not DCOM, although plans for DCOM support were announced in 1997.

Last but certainly not least, COM/DCOM it is also supported on Microsoft’s platforms, Windows 95, Windows 98 and Windows NT. It is at least theoretically possible to implement event services in COM covering those platforms. However, the runtime environments provided in non-Microsoft environments are not as extensive as in
the Windows environments, for instance, the ActiveX specifications are not fully supported in the EntireX runtime environments.

**Languages**

It is rather difficult to assess exactly which languages support COM at this point, as no official information on this could be found at the time of writing, but the following do support COM with certainty (this is not intended to be a complete list):

- C++, Visual Basic, Java, Borland Delphi (Object Pascal), SmallTalk, Sybase PowerBuilder and Oberon Microsystems Component Pascal.

**CORBA support**

**Platforms**

Supporting heterogeneous platforms has been one of CORBA’s design criteria from the start. However, it is up to individual companies to provide the ORB, and accompanying CORBA services. When this is written, there are at least three ORB providers competing for the CORBA market. These are Iona with its Orbix ORB, Bea systems with its ObjectBroker, and Inprise corporation (formerly Borland/Visigenic) with its VisiBroker ORB. Iona reports support for the following platforms [http://www.iona.com]: Sun Solaris, Hewlett Packard HP/UX, Silicon Graphics IRIX, Digital Unix, Digital Alpha/OpenVMS, Digital Alpha/ Windows NT, Sequent DYNIX and Microsoft Windows NT/95. Platform support was not available from Bea System’s web site at the time of writing. Inprise reports support for the following platforms [http://www.inprise.com/visibroker/]: Windows NT, Sun Solaris, IBM AIX, HPUX, SGI IRIX, Digital UNIX, IBM OS/390, Novell NetWare and vxWorks. Additionally, at least Iona Orbix and VisiBroker are available for Java, and thus accessible from all platforms supporting Java.
CORBA support is thus clearly available on a broad range of platforms.

Languages

Here the list is considerably shorter. The only language binding that is available over all platforms is C++. Java is also a language that can be used on many platforms as it is platform independent. Those are languages that can be used directly with CORBA. Other native language bindings are not supported by the leading ORBs to the best knowledge of the author. Additionally, where the ORB supports CORBA to COM bindings, languages that support COM can be used to develop CORBA compliant objects.

Summary

Both COM and CORBA support development over heterogeneous platforms/languages. Although the use COM for cross-platform development is possible, support for the technology on other platforms than Microsoft’s is limited. Cross platform CORBA support is more extensive, as it is supported on all major operating systems. COM appears to be supported by more languages/development environments, but the existence of CORBA to COM bindings make this picture a bit fuzzy.
6. Results

A quick summary of the results. An empty cell denotes “Neutral”, i.e. does not affect support for complex event detection. + means “Good” and - is “Bad”. This is a subjective assessment of the factors covered in the evaluation.

<table>
<thead>
<tr>
<th></th>
<th>CORBA &amp; Event Channels</th>
<th>COM &amp; Connection Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contracts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encapsulation</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Safety</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polymorphism</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Service level</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concurrency</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Semantics</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Heterogeneity</td>
<td>+Platforms -Languages</td>
<td>-Platforms +Languages</td>
</tr>
</tbody>
</table>

Contracts. The contract factor does not hamper either COM nor CORBA from supporting complex event detection. Versioning is required in COM, while the use of generic event channels will defeat versioning. However, it is not clear how complex event detection affects versioning of event sources. This is outside the scope of this dissertation, as it does not identify requirements for complex event detection that arise from the component context, it only considers the event model itself.
Polymorphism. COM events are not polymorphic by default, making a subset of existing event sources unusable for complex event detection, unless special adapters are implemented for such event sources. COM events using dispatch interfaces can be used. CORBA events can be handled polymorphically both when using the generic event channels and when using typed event channels, thanks to the Dynamic Skeleton Interface.

Encapsulation. CORBA provides the concept of event channels which is ideal to support encapsulation of events. COM users have to custom create encapsulation mechanisms for every (non-polymorphic) event source. CORBA restricts event parameter delivery in such a way that encapsulation of event parameters is not necessary. COM leaves open options that may endanger consistent reception of event parameters.

Concurrency. Neither COM nor CORBA define concurrent-safe multicast. Both provide concurrency support which can be used to implement concurrent complex event detection. CORBA defines a global time service, which COM lacks.

Semantics. This has to be left as an open issue as the specifications of COM and CORBA do not define any of the semantic requirements identified.

Safety. Event notifications can be statically type checked (except for the generic event channel) while event subscriptions cannot. Both offer run-time type checking that can help ensure safe event subscriptions. Neither specifies how to react to errors/exceptions while event notification is in progress.

Service level. As with semantics, this is an open issue as service level is not specified in neither case. CORBA event service designers claim that it is designed with various service factors in mind, while no such claims are made for COM connection points.
**Heterogeneity.** Both COM and CORBA are supported on more than one platform and by more than one language. The number of platforms available for CORBA development is greater, but the number of languages supporting COM is greater.
7. Conclusions and future work

7.1 Conclusions

In this dissertation, a view on events in component software has been established. Also, a view components is presented in the background chapter, and a definition is given. An evaluation framework based on the literature on components and events is defined, and it used to evaluate the suitability of two well-known event models to support complex event detection, the result of which is reported above.

It is the conclusion of the author that the hypothesis put forth in Aims has not been refuted, namely that the event models in question have to be improved in order to support advanced event services. This is mainly because of their lack of attention to the polymorphism factor, but also because of semantic and service level issues, neither specification defines the semantics or service level of the event models, which makes it difficult to build advanced event services on top of them. Also, the definition of the event notification mechanisms need to be restricted more in both models in order to ensure that an event defined using them will be usable for complex event detection. Both event models in question here can be improved in order to support complex event detection.

7.2 Limitations

Limitation of framework: The transaction dimension is not considered. Transactions influence the semantics of event models considerably, for instance it opens the question of what happens to event notifications that are done within a transaction which is then aborted.
Limitation of evaluation: The evaluation is done on the specification level. All results of these evaluations would also apply to respective implementations, but additional results could be obtained by performing an implementation level evaluation, especially when considering the factors of service level and semantics.

7.3 Future work

A project such as this often raises as many questions as it answers, and perhaps more. Issues not addressed include:

- Applying the framework to other component event models. One immediate candidate is JavaBeans with Java events.
- Applying the framework to implemented systems, such as specific ORBs.
- Versioning support in complex event detectors.
- Investigating the problem of ensuring system consistency while event delivery is taking place.
- Event models and module safety.
- Identifying further service level issues and their relation to event models.
- Determining the feasibility of an event model design that takes all the factors mentioned here into account.
- Extending the framework into the transaction domain.
8. Acknowledgements

I would like to thank Brian Lings for his enormous patience in supervising this work. I would have given up on myself a long time ago. Henrik Engström is largely to blame for the existence of this dissertation, my sincere thanks to him for his efforts. Thanks go to Jonas Mellin and Mikael Berndtsson for being willing to read this creature, and for their useful comments.

At last but certainly not least, I am grateful for the support of my family and friends who always believed in me, even when I didn’t.
## 9. Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBD</td>
<td>Component Based Development. The practice of developing applications by developing/buying components and combining them in an application.</td>
</tr>
<tr>
<td>DCE</td>
<td>Distributed Computing Environment. A standard for RPC created by the Open Software Foundation.</td>
</tr>
<tr>
<td>DII</td>
<td>Dynamic Invocation Interface. A specification in CORBA that allows object clients to invoke methods in a uniform way (polymorphically).</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic Link Library. A mechanism supported by many operating systems to enable dynamic extensions of programs and to share binary code.</td>
</tr>
<tr>
<td>DSI</td>
<td>Dynamic Skeleton Interface. A specification in CORBA that allows object servers to handle invocations in a uniform way (polymorphically).</td>
</tr>
<tr>
<td>JDK</td>
<td>Java Development Kit.</td>
</tr>
<tr>
<td>OCX</td>
<td>OLE Custom Control</td>
</tr>
<tr>
<td>Reflection</td>
<td>See RTTI</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>RTTI</td>
<td>Run-time type information. Information linked into an executable file that makes it possible to discover type information at run time. Also known as reflection/introspection.</td>
</tr>
<tr>
<td>VBX</td>
<td>Visual Basic eXtension. The component standard of Microsoft Visual Basic up to version 3</td>
</tr>
<tr>
<td>VCL</td>
<td>Visual Component Library. An acronym for the class library and component standard of Borland Delphi</td>
</tr>
</tbody>
</table>
10. Bibliography and References


Available on the WWW at http://www.omg.org

Available on the WWW at http://www.omg.org


[Gamma95] E. Gamma, R. Helm, R. Johnson & J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995


11. Appendix A. Overview of the evaluation framework

Contracts
- Event delivery contract
- Event parameters contract
- Event interface contracts
- Event versions

Polymorphism
- Polymorphic event notifications
- Polymorphic event subscriptions

Encapsulation
- Event source encapsulation
- Event parameter encapsulation

Concurrency
- Thread safe multicast
- Concurrent detection/signaling
- Concurrent event occurrences

Semantics
- Preservation of causality
- System state consistency
- Consistent multicast
- Fault tolerance
- Atomic event delivery

Safety
• Static type checking
• Run time type checking
• Exception handling

Service level
• Performance
• Accuracy

Heterogeneity
• Platforms
• Component models
• Languages
12. Appendix B. Selected listings.

12.1 Listing 1. A sample COM interface for passing events.

```c
// Interface ISnoopNotify
[
    object,
    uuid(32bb8324-b41b-11cf-d6ee-0080c6c45e45),
    helpstring("Snoop event notification interface"),
    pointer_default(unique)
]

interface ISnoopNotify : IUnknown
{
    // Pre: time_stamp contains the time of event occurence,
    //      variant type is VT_TYPE == DATE.
    //      event_type contains the name of the event.
    //      params contains user defined parameters.
    // Post: See guarantees made by the event service.
    HRESULT Notify([in] variant time_stamp,
                    [in] char* event_type,
                    [in] variant params);
}
```
12.2 Listing 2. The process of using COM Dispatch interfaces for passing events

1. Event service queries connection point container for its events
2. Event service advises the desired connection point to connect to its sink.
3. Connection point queries the event service to check if it supports the appropriate interface.
4. In the event services QueryInterface method, the event service checks if the requested event interface is a dispatch interface. If so, it returns success, otherwise it returns failure.
5. The event source can now notify the event service of occurring events using the Invoke method.

If the event source tries to notify the event service using a vtbl call, such a call will fail.