An Architecture for Viewpoint Environments based on OMG/CORBA

Wolfgang Emmerich
Dept. of Computer Science, City University
Northampton Square, London EC1V 0HB, U.K.
we@city.ac.uk

Abstract

One of the major goals of the Viewpoint framework is to allow for heterogeneous and distributed implementation of viewpoint-based tools. This poses a number of challenges on the integration of tools into a viewpoint-based environment. We review different infrastructures that could be deployed for tool integration and argue that the OMG/CORBA architecture provides the best support for achieving heterogeneity and distribution. We then discuss how viewpoint-based tools can be integrated on the basis of CORBA. This involves the integration of different viewpoint representations as well as their presentation at user interfaces.

1 Introduction

A number of different *tasks* have to be performed during the construction of a software-intensive system. Any such system of non-trivial size requires multiple developers with different knowledge and expertise to cooperate on the different tasks involved. Tasks range from requirements analysis over architectural and component interface design to component implementation, component test and integration test.

During the different tasks, the system is considered from multiple perspectives. Requirements analysis tasks tend to take different end-user perspectives and focus, for instance, on the functionality a system should offer from different users' point of view. Architectural design tasks take the perspective of an engineer who decides how the system should be constructed. Note that there are usually even multiple perspectives required during single tasks. Dardenne et al., for instance, suggest in [2] different requirements engineering perspectives for goals, actions, agents, constraints, entities and relationships. Developers have to realise these different perspectives for documentation and communication purposes in documents or artifacts, which we refer to as *viewpoints* so as to emphasise that they are different perspectives on the same system. We shall use the terminology established in [7].

2 Integrated Tools

Viewpoints are defined in a formal language as determined by the viewpoint's style slot. As different viewpoints represent different perspectives of the same system they are not fully independent of each other. There are rather multiple consistency constraints between the different viewpoints. A viewpoint defining a goal in a requirements specification, for instance, might use an entity definition and the use must match the entity declaration in some other viewpoint. This viewpoint might even be defined by some other developer. The large number of viewpoints in a system definition together with the various consistency constraints generate developers' demand for *tool support*. To be able to handle inter-viewpoint consistency constraints tools have to be integrated into *viewpoint-based environments*.

For each different viewpoint template, there will have to be a tool available, that enables developers to create instances of this template. Tool should offer *assembly actions* that create or modify the different viewpoint fragments as defined in the viewpoint's style slot. The tool will then have to accomplish *persistent storage* of viewpoints. It should also support developers in achieving viewpoint consistency by offering *in-viewpoint check actions* that check for syntactic and static semantic correctness of the viewpoint. As viewpoints are often related to other viewpoints, tools also have to support achieving inter-viewpoint consistency. This involves *inter-viewpoint consistency checks*, automated *consistency preservation actions*, such as change propagations, and management of temporarily tolerated interferences. Finally, the tool will have to take measures for *integrity preservation* of viewpoint representation against both, concurrent changes of multiple users and hardware or software failures.

An important concern during the definition of the viewpoint framework has been to allow for heterogeneity [7]. The initial *Viewer* [9] did not explore this aspect but was rather a homogeneous and centralised proof-of-concept prototype developed in Smalltalk. Heterogeneity includes the autonomy to choose whatever base technology is appropriate for the construction of viewpoint-based tools. Indeed, the first three of the above requirements, i.e. facilities for viewpoint assembly, viewpoint persistence and in-viewpoint checks can be dealt with in a fairly autonomous way. Viewpoint tool builders can choose a programming language and reuse appropriate frameworks for internal viewpoint representation. They may choose to store viewpoint representations in operating system files, as a Smalltalk image or in an object database. They can, finally, deploy different strategies for

implementing in-viewpoint check actions. This autonomy, however, cannot be granted for the other concerns.

To check for inter-viewpoint consistency constraints, one viewpoint-based tool will have to access the representation of related viewpoints that are maintained by some other viewpoint-based tool. Then integrity against concurrent updates of multiple users or hardware/software failures cannot be preserved locally, because they may involve changes to distributed tools. Hence, inter-viewpoint consistency checks and integrity preservation can only be addressed by an integration of multiple tools. The goal of allowing for autonomous viewpoint tool implementation seems to be contradictory to the integration of tools. We will, therefore, review integration infrastructures for the degree to which they support both.

3 Integration Infrastructures

3.1 Object Databases

In [6] we argued that a common database should be used as a tool integration infrastructure and database requirements were outlined that would facilitate this integration. An object database was extended to meet these requirements in the GOODSTEP project [8] and we have demonstrated in [4] how sophisticated tools can be constructed on top of the extended system. Tools in this environment support C++ class library design, implementation and documentation and the environment facilitates inter-viewpoint consistency checks and preservation, support of multiple concurrent users as well as version and configuration management. Their implementations utilise database features, such as object-oriented schema definition for fine-grained document representation, database transactions for concurrency control and a version manager for composite objects. The approach of relying on a single and centralised database, however, has a number of disadvantages that may restrict its applicability.

Firstly, databases are rather passive components. Hence they lack sufficient primitives for control integration. Object databases, for instance, do not provide a primitive that a tool, which has performed a change to a viewpoint, can use to inform concurrently running tools that would have to redisplay viewpoints to visualise the change. Secondly, the approach of using an object database for tool integration limits heterogeneity. All tool builders will have to use the same object definition language to define and implement their tools' schemas before schema export/import facilities can be used for tool integration purposes. Finally, a central object database server will become a performance bottleneck, if the number of concurrent transaction requests exceeds its transaction throughput capacity.

3.2 Message Router

To implement control integration, the object database tool architecture in [3] includes Sun ToolTalk, a message based tool integration framework. Through this integration infrastructure, tools can exchange parameterised messages and hence actively communicate with each other. This message passing can not only be used for implementing control integration, but also for implementing inter-viewpoint consistency checks between those tools that do not store viewpoints in an object database. Therefore, a tool T_1 would send a message, which is interpreted as a check request, to another tool T_2. T_2 would execute the check and send another message to T_1 transferring the result.

This message based approach to tool integration has several weaknesses, which are not confined to Sun ToolTalk but will also be found in other systems, such as the HP Broadcast Message Server. Firstly, the expressiveness of messages is fairly poor. Messages can be parameterised, but parameter types can only be of very limited number of atomic types, such as strings, integers and booleans. Secondly, there are no concepts for identifying similarities between different message types or expressing a certain behaviour of a message. Thirdly, message typing is not statically checked. This may lead to situations where one tool expects different parameters of a message than the message sending tool has provided. Fourthly, messages are sent asynchronously, but in many cases a sending tool would like to receive a response immediately. Finally, message based integration frameworks do not support concurrency control and also do not remove the performance bottleneck of a central database server.

3.3 OMG/CORBA

These problems are reasonably well solved in OMG's common object request broker architecture (CORBA) [10]. Figure 1 displays an overview. The core component of this architecture is an object request broker (ORB), which is specified in detail in [11]. An ORB delivers a client's request for invoking an operation to a server object, which will execute the operation and then the ORB will deliver the result to the client. Objects have a type, which is defined in the OMG interface definition language (IDL). IDL supports all main-stream object-oriented concepts, such as encapsulation, multiple inheritance and polymorphism. Client and server objects can be distributed and heterogeneous. They need not be running on the same machine or the same operating system, and need not be implemented in the same programming language. CORBA object requests are synchronous, though asynchronicity is supported by the event notification service (see below). The CORBA specification identifies a large number of failures that can occur during the processing of an event and ORBs are supposed to detect those failures and notify clients that exceptional situations have occurred. In addition to general failures, type specific exceptions can be specified in IDL.

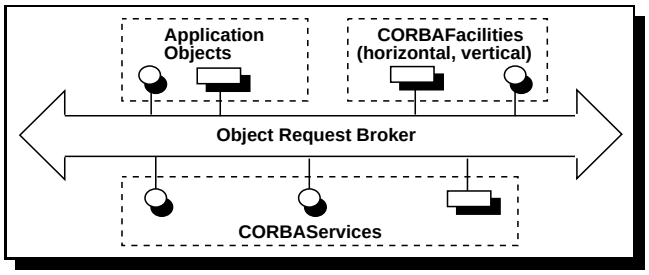

Figure 1: OMG CORBA

A number of fundamental problems that are common to any heterogeneous and distributed computing architecture are addressed as CORBAServices. Among others these include the definition of external names to objects (Naming), distribution of events among multiple anonymous objects (Event Notification), maintaining relationships between objects and forging them to composite objects (Relationship),

the creation, migration and removal of objects (Lifecycle), persistent storage of objects (Persistence), subsumption of multiple object requests to atomic transactions (Transaction), and object concurrency control (Concurrency). The services have IDL interfaces and are available as CORBA objects.

Layered on top of the services, the architecture defines CORBAFacilities. These are components that are useful horizontally across domain borders or specific for certain vertical market segments. The first horizontal facility adopted by the OMG is the distributed document component facility that evolved from the OpenDoc Framework produced by Apple, IBM, CI Labs and Novell. It features document components and their composition into compound documents as well as their portable visualisation across various user interface platforms.

To utilise OMG/CORBA as a tool integration infrastructure, viewpoint-based tools, viewpoints as well as viewpoint components have to be considered as CORBA Application Objects. Therefore, their types have to be defined as OMG/IDL interfaces and their implementations have to be registered with an object request broker. The operations exported by viewpoint-based tools would be defined in IDL interfaces. A viewpoint-based tool object, for instance, would offer operations to instantiate the viewpoint template associated with the tool so as to create a new viewpoint. Likewise, viewpoint objects would offer operations that implement the work plan and eventually fill the specification slot. Invocation of these operations could then be requested by any other client object belonging to the viewpoint-based environment. As an example consider Figure 2, which includes IDL declarations for a viewpoint-based tool for Entity definitions as they would occur in a goal-oriented requirements analysis environment.

```
#include "VP.idl"
module VPEntity {
 exception LexicalError{short pos};
 exception NameAlreadyDefined{Entity defined_in};
 exception NameDoesNotExist;

 interface EntityName : VP::ViewpontComponent {
  readonly attribute string value;
  void Scan(in string value)
   raises(LexicalError);
 };

 interface Entity : VP::Viewpoint {
  readonly attribute EntityName name;
  void Expand();
  ...
 };

 interface EntityEditor : VP::ViewpointTool {
  Entity CreateEntity(in string name)
         raises(NameAlreadyDefined);
  Entity FindEntity(in string name)
         raises(NameDoesNotExist);
  ...
 };
}
```

Figure 2: IDL Examples

The example outlines three interface definitions. Interface `EntityEditor` defines a viewpoint-based tool for defining entities. It provides an operation to create a new entity and another operation to find an entity by name. `Entity` is the interface for an entity viewpoint. We assume that a name has to be defined for an entity, which is stored in an attribute that is readable. `Entity` defines an operation `Expand` that can be used to create a new name object and store it in the attribute. `EntityName` has an attribute of type string where entity names are stored. It exports an operation that scans the attribute name for lexical correctness. User-defined exceptions are used for error handling.

The interface definitions are defined within an IDL module `VPEntity`. This module defines the scope for declarations and avoids name clashes with other modules defining other viewpoint-based tools. Thus modules ensure the autonomy of tool builders to choose appropriate identifiers without having to worry about name clashes with the choices of other tool builders. The interfaces are derived from pre-defined interfaces that define common properties of viewpoint-based tools, viewpoints and viewpoint components. These interfaces are defined in module `VP`, which is defined in file `VP.idl`. It is included to make its declarations known in module `VPEntity`.

The benefit of using CORBA to invoke an operation, as opposed to a local procedure call is that the client object need not be implemented in the same programming language, it need not be running in the same process, it need not run on the same operating system and can be distributed over a local-area or even a wide-area network. The object need not even know the physical location of the object it requests the service from, but only needs to possess a reference on the object. In the example above a viewpoint-based tool for goals, for instance, could check whether an entity that is used in a goal definition exists by invoking operation `FindEntity`. If the invocation reveals an exception, it can create the entity. The two viewpoint-based tools, however, can be implemented in completely different languages and be running on heterogeneous hardware platforms, but will still be interoperable. For invoking these operations, the tool for goal viewpoints would only have to possess a reference of the object of type `EntityEditor`.

Naming: The principle way for a client object to obtain an object reference of a viewpoint-based tool is to use the naming service. Tools, therefore, register themselves with the naming service and create a name binding between an external tool name and the CORBA object implementing the tool. Clients that wish to invoke an operation from a remote tool can then obtain an object reference by submitting the tool's external name to the naming service. In the above example, a name binding of an external name and the instance of type `EntityEditor` had to be created. Then the goal viewpoint tool could resolve the name binding by submitting the external name to the naming service and obtain the desired object reference.

Relationships: The CORBA relationship service provides the foundation to manage different relationships between viewpoint components. To create a relationship between two viewpoint components, neither the interfaces of the viewpoint components nor the component objects themselves have to be changed. Hence relationships can be added and removed in a fairly autonomous fashion. The relationship service distinguishes aggregation relationships and reference relationships. Aggregation relationships can be used to de-

fine the components of a viewpoint. Reference relationship are used to refer in one viewpoint component to other components of the same or other viewpoints. In the above example, a goal that uses an entity can have a reference relationship that can be exploited by a static analyser or to implement efficient change propagations. The use of the relationship service also enables viewpoints to be treated as composite objects by the lifecycle service. Then the lifecycle service can be used to implement tool replication or migration to a different hardware platform or viewpoint deletion.

Event Notification: The CORBA event notification service enables the exchange of events between different tools. An event may be, for instance, that a viewpoint has been changed. Objects can register themselves with an event channel and declare interest in all events the channel will be notified about. A tool may then notify the event channel of an event occurrence and the channel would multi-cast the event to interested parties. Asynchronous communication between viewpoint-based tools is achieved by a pair of event channels, one for sending requests and another one for sending results.

Persistence: CORBA persistence can be exploited for storing viewpoints and viewpoint-based tools persistently. The service manages the storage of a CORBA object's state in some datastore. The service supports different types of datastores by three protocols. The direct access (DA) protocol can be used with relational databases, the ODMG-93 protocol defines how object states are stored in ODMG compliant object databases and a generic dynamic data object (DDO) protocol can be used with any kind of data store, for instance the file system. Hence, this service supports autonomy of viewpoint owners in choosing the appropriate database for storage of viewpoints. The essence of these protocol definitions are languages that are used to define those attributes of an object whose states have to be stored persistently. These cannot be inferred from the interface definition because there may be hidden attributes that are not exposed at the interface or exported attributes may not have to be stored persistently. The protocols, therefore, either define or reuse data definition languages (DDLs). The DA protocol defines a DDL, which is a proper subset of IDL and the ODMG-93 protocol reuses ODL, which is the schema definition language of the ODMG database standard [1]. The persistence service implementation is supposed to provide a compiler that generates the required code for actually storing and retrieving attribute values. Tool builders can exploit the ODMG-93 protocol by using ODL to define the set of attributes that need to be stored. Then an ODL compiler would generate an object database schema for these types as well as code fragments that transfer attribute values from CORBA objects into database objects and vice versa.

Concurrency: Viewpoint-based tools may chose to store viewpoints in different types of databases. Concurrency control can, therefore, not be achieved by the concurrency control manager of a single database system. Instead a federated concurrency control management is required. The CORBA concurrency control service provides the primitives for that by defining an interface for locksets. Locksets manage the set of locks that were granted to different concurrent threads or processes. If locksets are associated to viewpoints or individual viewpoint components locksets can be used to restrict concurrent viewpoint access so that viewpoint integrity is preserved. To achieve that, operations exported by locksets have to be used to acquire, upgrade or release locks on viewpoints or viewpoint components. The service defines different lock modes (such as `read` and `write`) and a lock compatibility matrix such that a lock is only granted if it does not conflict with any lock that has been granted before. The lock compatibility matrix is defined in a way that lost updates or inconsistent analysis problems cannot be caused by concurrent object accesses. The service is also prepared to be used by the CORBA transaction service.

Transactions: Atomicity of a sequence of operation invocation requests, such as changes to a set of related viewpoints, may not be achievable by database transactions if the requests involve changes to viewpoint components stored in different database systems. It can also not be achieved if viewpoints are stored in file systems, which do not have a transaction management at all. Therefore, other means for facilitating atomicity have to be provided. The CORBA transaction service provides a framework for implementing a two-phase commit protocol which features atomicity of distributed and possibly nested transactions. Viewpoint-based tools can ask a transaction coordinator to start and commit transactions. The effect of all operation requests executed between begin and commit will then be atomic, i.e. it will be visible completely if the commit is reached or not at all if a failure occurred or the transaction was explicitly aborted. Transaction coordinators also isolate transactions from each other. The coordinators, therefore, implicitly use the concurrency control service. Every object that is accessed or modified during a transaction will be locked by the transaction coordinator, and the coordinator will release all locks during commit or abort.

4 User Interfaces and CORBA

The review above has clearly indicated that CORBA provides a useful infrastructure for integrating viewpoint-based tools in a heterogeneous and distributed environment supporting multiple developers. Now the question arises how tool's user interfaces can be hooked into this infrastructure. We sketch four different approaches to user interface construction and discuss for each of those how operations defined in viewpoint interfaces can be invoked.

User interfaces on UNIX platforms are built either by reusing user interface management libraries, such as ET++ or Interviews, or by using a GUI builder that provides a graphic editor to layout the user interface and then generates code fragments in Smalltalk, C or C++ to be filled in with the application code. If viewpoint-based tools are constructed following these approaches, interactions between users and tools are implemented in Smalltalk, C or C++. The IDL language bindings to Smalltalk, C and C++ can be used for interfacing with CORBA objects implementing the viewpoints functionality. The language bindings are implemented in a static way by the IDL compiler. It generates static invocation stubs in Smalltalk, C or C++. The user interface can then invoke viewpoint operations that were specified in IDL as if they were C, Smalltalk or C++ operations.

The distributed document component facility (DDCF) proposed by IBM and Apple seem to be very appropriate for constructing user interfaces of viewpoint-based tools. Hav-

ing evolved from OpenDoc, DDCF supports the composition of complex documents from simpler parts. These parts are being created and edited by component applications. The DDCF could be deployed for a viewpoint-based tool by deriving viewpoint types from the DDCF type for root parts and types for viewpoint components from DDCF embedded parts. Then viewpoints and viewpoint components would inherit operations provided by the respective OpenDoc applications and OpenDoc would also take care of displaying them. Operations specific for viewpoints, such as viewpoint consistency checks could be added in the viewpoint specific subtypes. Being a specialised CORBA facility, viewpoints could then still be distributed and use the CORBA services discussed above.

A portable way to construct user interfaces is to use Java. The Java awt class library provides all primitives for user interface construction and Java programming environments are available that provide graphical user interface editors. Although the OMG has not accepted a standardised Java binding for IDL yet, there are several ORB products that enable Java applets to communicate with other CORBA objects. Hence a viewpoint-based tool could have a user interface implemented in Java and running in any Java enabled Web browser. The user interface would then use a Java/IDL binding to invoke operations that implement the viewpoint's functionality via an object request broker.

User interfaces for viewpoint-based tools on Windows PCs would likely to be constructed using Microsoft's OLE. The OMG recently adopted the first part of a specification for OLE/COM interworking, which includes a mapping between the two object models and between OLE and CORBA. With the second part to be completed later this year, it will be standardised how operations of CORBA objects are invoked from OLE objects and vice versa. This will facilitate user interfaces of viewpoint-based tools running on PCs while the viewpoint functionality is implemented in CORBA objects.

5 Further Work

Although we have done some experimentation with ORB products, we do not yet have a clear understanding of the feasibility of the approach sketched here from a performance point of view. We plan to develop a benchmark that represents a synthetic load that would be put on an ORB by a viewpoint-based environment. We believe that most of the techniques for developing dedicated object database benchmarks presented in [5] can also be applied to a distributed computing environment. The benchmark will then be used to compare the performance of a number of object request broker products and to see whether the response times are sufficient for using ORBs within a viewpoint-based environment for which we want to develop a proof of concept prototype following the architecture outlined here.

To date, there are about ten object request broker products in the market. None of them offers all the services that have been adopted. An important observation is that the persistence service is not implemented in a compliant way by a single product. Also the concurrency control and transactions services are not yet widely available. We are currently studying the cause of this problem and may attempt an implementation of these services with appropriate partners.

References

[1] R. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufman, 1993.

[2] A. Dardenne, A. van Lamswerde, and S. Fickas. Goal-directed Requirements Acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[3] W. Emmerich. *Tool Construction for process-centred Software Development Environments based on Object Database Systems*. PhD thesis, University of Paderborn, Germany, 1995.

[4] W. Emmerich, J. Arlow, J. Madec, and M. Phoenix. Tool Construction for the British Airways SEE with the O_2 ODBMS. Technical Report 1996/01, ISSN 1364-4009, City University London, Dept. of Computer Science, 1996.

[5] W. Emmerich and W. Schäfer. Dedicated Object Management Benchmarks for Software Engineering Applications. In R. Welland, editor, *Proc. of the Software Engineering Environments '93, Reading, UK*, pages 130–142. IEEE Computer Society Press, 1993.

[6] W. Emmerich, W. Schäfer, and J. Welsh. Databases for Software Engineering Environments — The Goal has not yet been attained. In I. Sommerville and M. Paul, editors, *Software Engineering ESEC '93 — Proc. of the* 4^{th} *European Software Engineering Conference, Garmisch-Partenkirchen, Germany*, volume 717 of *Lecture Notes in Computer Science*, pages 145–162. Springer, 1993.

[7] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: a framework for integrating multiple perspectives in system development. *Int. Journal of Software Engineering and Knowledge Engineering*, 2(1):21–58, 1992.

[8] GOODSTEP Team. The GOODSTEP Project: General Object-Oriented Database for Software Engineering Processes. In K. Ohmaki, editor, *Proc. of the Asia-Pacific Software Engineering Conference, Tokyo, Japan*, pages 410–420. IEEE Computer Society Press, 1994.

[9] B. Nuseibeh and A. Finkelstein. ViewPoints: A Vehicle for Method and Tool Integration. In *Proceedings of the* 5^{th} *Int. Workshop on Computer-Aided Software Engineering (CASE '92), Montreal, Canada*, pages 50–60. IEEE Computer Society, 1992.

[10] R. M Soley, editor. Object Management Architecture Guide. Technical report, Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701, USA, 1992.

[11] R. M Soley, editor. The Common Object Request Broker: Architecture and Specification Revision 2.0. Technical report, Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701, USA, July 1995.