CORBA and ODBMSs in Viewpoint Development Environment Architectures

Wolfgang Emmerich
Interoperable Systems Research Centre, City University
Northampton Square, London EC1V 0HB, UK

Abstract

Viewpoints are reflections of software systems from multiple perspectives. A number of consistency conditions apply to viewpoints and developers require a tool for each type of viewpoint. These tools need to support consistency management. Inter-viewpoint consistency can only be checked when tools are integrated into a viewpoint development environment. We briefly outline the functionality developers require from these environments. We discuss the suitability of abstract syntax graphs as a common viewpoint representation scheme. The main purpose of the paper is to present an object-oriented architecture for viewpoint-based environments. The architecture benefits from the integration of object database management systems and object request brokers.

1 Introduction

The production process of modern software systems passes many stages, such as requirements analysis, architectural design, detailed design, coding, testing. During these stages, the system is considered from multiple perspectives. Requirements analysis tends 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 takes the perspective of an engineer who decides how the system should be constructed. These perspectives highlight the principal system components and their interaction mechanisms. Note, that multiple perspectives may even be required during a single stage. Dardenne et al., for instance, suggest in [4] different requirements engineering perspectives for goals, actions, agents, constraints, entities and relationships. Developers have to materialise those different perspectives for documentation and communication purposes in documents or artifacts, which we refer to as *viewpoints* [14] so as to emphasise that they provide different perspectives on the same system. This viewpoint-oriented perspective on software processes is shared by an increasing number of authors (see [17]).

Viewpoints are defined as a loosely-coupled, locally managed objects encapsulating representation knowledge, development process knowledge and partial specification knowledge about a system. Viewpoints are structured into five slots: The *style* slot describes the representation scheme used by the viewpoint. The *work plan* slot describes development actions together with a strategy for

their application to construct the viewpoint. The *domain* slot identifies the viewpoint and sets its context in the overall system under construction. The *specification* slot describes the viewpoint domain in the representation scheme determined by the style slot. The *work record* slot records the history and current state of the viewpoint development.

Viewpoints are defined in a formal language, determined by the viewpoint's style slot. As different viewpoints represent different perspectives on the same system they are not fully independent of each other; there are multiple consistency constraints between the different viewpoints. A viewpoint defining a goal in a requirements specification, for instance, might use an entity definition and this use must match the entity declaration in some other viewpoint. The high number of viewpoints likely in a system definition, together with the various consistency constraints, generate a demand for integrated *tool support*. Tools should, in particular, provide for intra- and inter-viewpoint consistency checks and manage temporary inconsistencies. Intra-viewpoint checks should reveal syntactic and static semantic errors within the viewpoint, while inter-viewpoint consistency checks should reveal inconsistencies between different related viewpoints.

Section 2 is devoted to a discussion of the functionality required from viewpoint-based tools in order to apply viewpoint-based methods effectively. In Section 3, we discuss an architecture for the integration of multiple tools into an environment that enables multiple developers to cooperate while using such methods. We particularly focus on the *a posteriori* integration of autonomously constructed tools. In Section 4, we discuss related work and outline open problems that need further attention in Section 5.

2 Viewpoint-based Environments

To apply a viewpoint-based method effectively, developers need a tool for each viewpoint template. Developers require particular functionality from each such tool. Some functions, however, cannot be provided by a single tool, but require the integration of tools associated with different related viewpoints. We refer to a set of these integrated tools as a *viewpoint-based development environment.*

A single tool enables developers as well as other tools to instantiate a viewpoint template and to complete the contents of the viewpoint, that is the specification slots. Therefore, the tool should offer *editing commands* for all the assembly actions identified in the work plan of the viewpoint template.

Inter-viewpoint consistency constraints imply relationships between different fragments contained in different viewpoints. It is often important for a developer to be able to traverse these relationships efficiently. A requirements engineer who is concerned with requirements traceability, for instance, may want to review the class in an object-oriented design that refines an entity definition. The viewpoint tool for entity definitions should, for example, offer a browsing command that makes a Booch viewpoint tool highlight the class definition refining the entity. The Booch viewpoint tool, in turn, may offer a

browsing command to find the C++ interface of the class. Note, that already this simple browsing facility requires viewpoint tools to be integrated.

Support for assembly actions and browsing facilities does not necessarily require that viewpoint-based tools be constructed. They could equally well be supported by generic text/graphic editors and hypertext viewers. The support that distinguishes a viewpoint-based tool from a generic editor is *consistency handling*. With respect to tool support, there are several facets of consistency handling. Consistency checks may be applied in a *lazy* or *eager* way. In a lazy consistency checking approach developers of a viewpoint will decide when to perform a check and what constraints to check for. In an eager approach the viewpoint-based tool checks for consistency while the user modifies a viewpoint and provides immediate consistency feedback. Observed inconsistencies may be *tolerated* or *prohibited*. Tolerated inconsistencies must be managed and different *resolution strategies* may or may not be enforced at some stage of the viewpoint development.

If multiple developers cooperate on the resolution of inconsistencies, it will inevitably be necessary to review the impact of everyone's changes as they occur. With an eager approach towards constraint checking, the visualisation of inconsistencies in one developer's viewpoint should be removed as soon as some other developer has removed the source of the inconsistency. We note that viewpoint-based tools have to be integrated in a way that they can access and update each other's viewpoints in a concurrent way.

It is often not appropriate for a developer to be disturbed by other developers' changes. Developers may want to work in isolation for some period of time, especially when they perform major changes to a viewpoint. Moreover, their changes should only become visible to other developers after they have reached a certain degree of (in-viewpoint) consistency. A way of achieving this is to arrange for viewpoint-based tools that are able to maintain different *versions* of a viewpoint. Version management has so far only gained widespread attention for source code viewpoints that are produced during the implementation stage, but we strongly believe that any viewpoint produced during any stage of a software process deserves the same attention. The concept of viewpoint versions is not only required not only to isolate developers from each other, but also to keep track of viewpoint history while a system is under maintenance. When a system is ported to a new platform, for instance, the versions of viewpoints for the previous platform must be retained. Developers will then need to *freeze* versions of a viewpoint so as to prevent it from being further modified. This will be necessary whenever a viewpoint has reached a state to which it might have to be restored in the future. Developers will then need a mechanism to *derive* a version from another frozen version and *select* a particular version. Then successive changes must only be done in that selected version. If no version is selected, a *default version* of a viewpoint will be used. Further version management support is required for labelling versions, traversing through the version history graph and for merging different alternatives into a common successor version.

3 Environment Architecture

Although viewpoints result from the different perspectives of different agents, it will be essential to have a common conceptual representation for all viewpoints so as to facilitate the definition of inter-viewpoint check actions. Having a common conceptual representation, however, does not imply that all viewpoints are supported in a uniform way and stored centrally.

3.1 Conceptual Viewpoint Representation

The style slots of viewpoints determine formal graphical or textual languages. Even viewpoints that express informal perspectives may have a structure of, for example chapters and sections, that can be expressed in a mark-up language such as SGML [12]. In search of a common conceptual viewpoint representation, we can, therefore, assume that any viewpoint has some form of syntactic structure that can be exploited.

The definitions of inter-viewpoint consistency constraints make reference to syntactic viewpoint fragments [9]. If a consistency condition relates two fragments, it is highly beneficial to materialise this relationship within the viewpoint representation, as opposed to computing it whenever the relationship is needed; the relationship should be materialised as soon as the related fragments are consistent with each other. Change propagations and browsing operations can then be implemented fairly efficiently by traversing the relationship. If one or the other related fragment is changed, relationships can be exploited for incremental reevaluation during check actions. Fragments to be related are often lexemes that match terminal symbols of the underlying grammar rather than more coarse-grained morphemes. Given that lexemes are usually involved in relationships, it will be necessary to model the complete syntactic structure of viewpoints using the common conceptual representation.

To break down the syntactic structure of a viewpoint, the specification slot of each viewpoint can be represented as an abstract syntax tree (AST), whose structure is determined by the grammar in the style slot. Nodes in this AST represent morphemes and lexemes. Nodes that represent morphemes are generated by productions of the viewpoint grammar and are referred to as *non-terminal nodes*. Lexemes are represented as *terminal nodes*, which have been matched with terminal symbols of the viewpoint grammar. Edges in these ASTs represent the syntactic decomposition and are referred to as *syntactic edges*. Nodes may have attributes for storing layout information or semantic data, for example symbol tables or details about inconsistencies to be resolved.

ASTs are generalised to abstract syntax graphs (ASGs) by the introduction of *semantic edges*. Such an edge between two nodes represents a semantic relationships between the two lexemes or morphemes represented by the nodes. Note that these edges lead to nodes contained in ASTs of other viewpoints if they are due to an inter-viewpoint consistency constraint. Therefore, the representation of all viewpoints that are produced in a project can be considered as one *project-wide abstract syntax* graph (see [8, 5]). The distinction between

syntactic and semantic edges achieves the identification of those *subgraphs* that model single viewpoints. All nodes that are in the transitive closure of nodes reachable via syntactic edges from a designated root node belong to the viewpoint together with all semantic edges connecting nodes within that set. These edges model semantic relationships due to in-viewpoint consistency constraints.

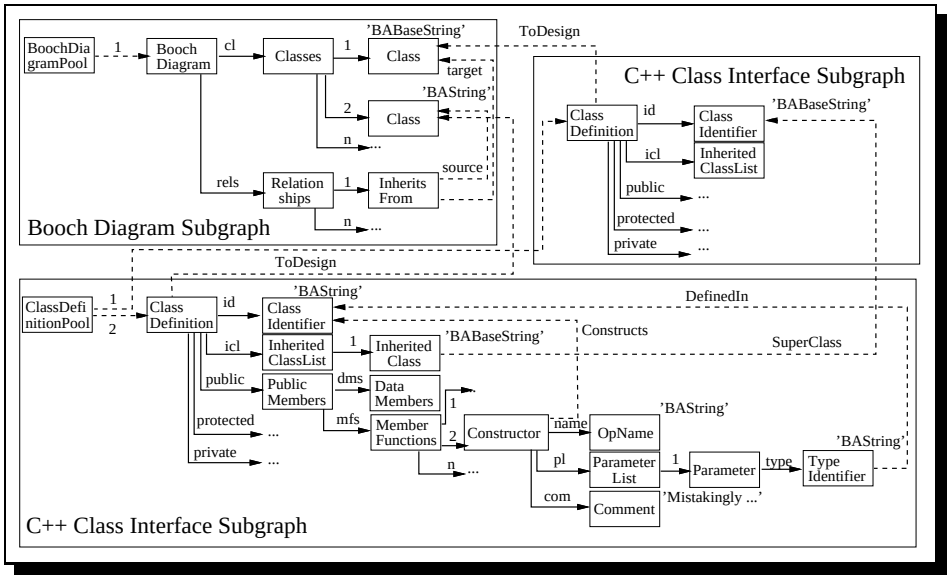

Figure 1: Fragment of a Project-wide Abstract Syntax Graph

As an example, consider Figure 1, which shows three excerpts from viewpoint subgraphs. The subgraph in the upper left corner represents a graphical Booch diagram viewpoint. The subgraph below represents an interface for a C++ class `BAString` that occurs in the Booch diagram and the subgraph in the upper right represents the class interface of `BABaseString`, the super class of `BAString`. Node attributes, given in quotes at the upper right corner of a node representation, store lexemes. Syntactic edges are drawn with solid arrows and semantic edges are displayed as dashed arrows. The edge `ToDesign`, for instance relates the root node of a C++ class interface viewpoint subgraph to the node in the Booch diagram viewpoint subgraph that designs the class. Although edges are directed, they are considered traversable in both directions. The two nodes labelled `BoochDiagramPool` and `ClassDefinitionPool` act as directories for the two viewpoint templates involved and are the starting point for queries that have to look-up a particular viewpoint that has been instantiated from that template.

This representation scheme is particularly appropriate for the efficient implementation of actions identified in a viewpoint template's workplan. Assembly actions are merely replacements of AST subtrees. In- and intra-viewpoint check action can be implemented on the basis of traversals through the project-

wide ASGs. Semantic edges considerably reduce search spaces and shorten traversal paths. Checking, for instance, the inter-viewpoint consistency constraint that each inheritance link in a Booch diagram viewpoint should be reflected by a respective declaration in the C++ class interface viewpoint can be implemented in the Booch diagram in the following way: For each `InheritsFrom` node in a Booch diagram viewpoint subgraph, an edge `target` leads to a superclass c_s of a class c identified by the `source` edge. The `ToDesign` edges leading to c_s and c can be traversed in reverse direction to find the root nodes c'_s and c' of the C++ class interface subgraph refining c_s and c. Then the `icl` edge starting from c' is used to find the list of inherited classes. After that the list is traversed and searched for a node whose lexeme attribute equals the lexeme attribute of c_s. Finally, a new semantic edge is created to connect the found `InheritedClass` node with the class identifier node that is reached by traversing edge `id` from c'_s. This edge might then be exploited for change propagations of the class name of c'_s or for efficiently browsing to all subclasses of c'_s.

3.2 Autonomous Viewpoint Implementation

The question arises how abstract syntax graphs can be implemented in an *autonomous* manner. Autonomy is particularly important in order to achieve interoperability of tools, which might not be changeable if they have been bought off-the-shelf. Then the abstract syntax graph structures and operations have to be implemented using the integration capabilities that off-the-shelf tools offer. Two major problems have to be solved to accomplish autonomy: heterogeneity and distribution. Heterogeneity of subgraph implementation occurs since different tools will use different hardware- and operating system platforms. It is also quite likely that different programming languages have been used for implementing tools. As different hardware platforms use different data representation formats and as different programming languages might be incompatible with each other, an integration framework is needed to resolve such heterogeneity. Distribution is required for both the type and instance level implementation of project-wide ASGs. At a type level, viewpoint owners have to define the structure and operations of ASG subgraphs to represent their viewpoint templates. At an instance level, subgraphs have to be distributed so as to meet the performance expectations of developers. A centralised ASG database server might become a performance bottleneck when serving a large numbers of users. Therefore, distributed ASG servers are needed for different viewpoint subgraphs to overcome such a performance bottleneck.

An architecture that has faced the challenge of supporting distributed and heterogeneous computation is OMG's CORBA [16]. To use a CORBA implementation for achieving a distributed and heterogeneous implementation of viewpoints effectively, project-wide ASGs have to be defined using the object-oriented concepts supported in CORBA's interface definition language (IDL). In [6] we have discussed how ASGs can be defined with object-oriented concepts. These considerations also apply to IDL. The main idea is to use classes to define ASG node types. Attributes model both node attributes and edges.

Edges are modelled as a pair of attributes so as to allow for traversals in both directions. Attribute types restrict the type of nodes to which edges may lead. Inheritance is used to model common properties of node types. Heterogeneous edges, i.e. edges that may lead to different types of nodes, are modelled by polymorphism. ASG operations that represent workplan actions are modelled as operations of classes. We suggest the use of IDL modules for restricting the name space of definitions such as node type declarations.

```
module BoochDiagrams {
 interface Class;
};

module ClassInterfaces {
 interface ClassDefinition:NonTerminal{
  readonly ClassIdentifier id;
  readonly InheritedClassList icl;
  readonly Members public;
  readonly Members protected;
  readonly Members private;
  readonly
    BoochDiagrams::Class ToDesign;
  ...
 };
 interface ClassIdentifier:Terminal {
  readonly CORBA::String value;
  readonly InheritedClass INVDefinedIn;
  void ChangeId(in CORBA::String new);
  ...
 };
 ...
};
```

```
module ClassInterfaces {
 interface ClassDefinition;
};

module BoochDiagrams {
 interface Classes : NonTerminal {
  readonly sequence<Class> l;
  Class addClass();
  void deleteClass(in Class cursor);
  ...
 };


 interface Class : Terminal {
  readonly CORBA::String value;
  readonly InheritsFrom INVtarget;
  readonly ClassInterface::
         ClassDefinition INVToDesign;
  void ChangeId(in CORBA::String new);
  ...
 };
 ...
};
```

Figure 2: ASG definition in IDL

As an example, consider the excerpts from two IDL fragments shown in Figure 2. The definitions on the left-hand side model nodes of the C++ class interface subgraph displayed in Figure 1 and the definitions on the right-hand side model nodes of the Booch class diagram subgraph. Let us now review how the semantic edge `ToDesign` that leads from the C++ class interface subgraph into the Booch diagram subgraph is modelled. The class interface implementation contains a forward declaration, an import that declares the existence of a module `BoochDiagrams` with a class `Class`. This import enables the qualified type `BoochDiagrams::Class` to be used in the definition of class `ClassDefinition` as an attribute type for `ToDesign`. The reverse direction of the edge `ToDesign` is modelled by attribute `INVToDesign` in class `Class` of module `BoochDiagrams`. Attributes are declared here as read-only, which means that the edges they model can be traversed, but not modified by any other means than the operations defined for the class. The operation `ChangeIdent` in class `Class`, for instance, might change the name of a class in a `BoochDiagram` and then implement a change propagation to the related C++ class interface definition viewpoint that is reached by following the `INVToDesign` attribute.

It is not necessary to model all node types of a subgraph in IDL, because

only those nodes that participate in inter-subgraph semantic edges have to be accessed from heterogeneous and distributed viewpoint-based tools. Therefore, only these have to be exposed in an IDL interface. In-subgraph traversal paths starting from these node types can be implemented in terms of local operations. This considerably increases performance for in-subgraph traversals and modifications, since local rather than remote operation invocations are used.

3.3 ODBMSs for Persistent Viewpoint Storage

Some viewpoint tools may be bought off-the-shelf, others may be constructed anew. In this subsection we sketch how object database management systems can be employed for the construction of new tools.

Given that CORBA is not required for in-subgraph traversals and operations, subgraphs can be stored in an object database management system (ODBMS) in the way suggested in [7]. A review of current database technology that lead to the selection of ODBMSs is provided in [2]. ODBMSs have been standardised by the object database management group [3], a group associated with the OMG. ODBMSs combine object-oriented programming languages with database technology. ODBMSs that comply to this standard offer an object definition language (ODL) for schema definition purposes. ODL is a strict superset of IDL. Database technology is used to transparently manage (1) the mapping of objects to secondary storage media, (2) integrity preservation of objects against system failures, (3) control of concurrent access to these objects and (4) network access from multiple clients to a central object server.

Objects managed by ODBMSs are persistent if they are reachable from a *persistent root*. Persistent roots are designated in the schema. To store subgraphs of a project-wide ASG persistently, we declare a set as a persistent root for each type of root node of each subgraph. As soon as we enter an object that implements a root node into this set the object becomes persistent. As the subgraph is spanned by a tree of syntactic edges, each node in the subgraph is reachable from the root node and therefore it is also persistent.

To meet the requirement of version management, multiple versions of those subgraphs that represent viewpoints have to be managed. Since these subgraphs are implemented by multiple rather than single objects, version management of collections of objects has to be supported. Mechanisms for that have not (yet) been addressed in the ODMG standard, though there are object databases that provide version management facilities for collections of objects.

O_2 [1], an ODMG compliant ODBMS, has been extended with such version management facilities, as discussed in-depth in [6]. The facilities are offered as a pre-defined class `o2_version` that maintains a collection of objects that are versioned together. The class then provides methods to add or delete objects from this collection and maintains a version history graph for it. It provides operations to traverse through this graph, to derive new version, to compare the differences between versions, to merge versions, to store a default version and to select a version other than the default version. O_2 manages a lazy object duplication strategy, which means that different versions of a version

collection share versions of component objects as long as they do not differ. For an evaluation of this duplication technique, see [6].

This versioning facility can be exploited during viewpoint-based tool construction for version management of subgraphs stored in an O_2 ODB. Thus, an instance of class `o2_version` is associated with the root node of each subgraph. New objects implementing new nodes of the graph are inserted into the version collection associated with the root of the graph. Then version management operations for subgraphs can be implemented directly on the basis of methods exported from `o2_version`.

ODBMSs support the concept of ACID transactions that protect the integrity of ASG subgraphs against hardware- or software failures. They also ensure serialisability of concurrent object updates. This will avoid inconsistent analysis and lost update problems that can occur during concurrent ASG updates. Viewpoint-based tools use these transactions to implement single assembly or check actions, rather than complete editing sessions. The duration of these actions tend to be in the order of magnitude of several hundred milliseconds rather than hours or days. The advantage of using commands rather than sessions as the unit of concurrency control is that other developers see the effect of a modification immediately after the action has been completed. As discussed in [6], computer supported cooperative work can be achieved in this way.

3.4 Combining CORBA and ODBMSs

Assuming that viewpoint development environments might contain external tools that do not store viewpoints as objects within an ODBMS, the need arises for those tools to use an object request broker to access ODBMS objects. This means that ODBMS objects have to implement IDL object interfaces. Vice versa, tool operations of ODBMS based tools are implemented within an ODBMS schema. They might have to access viewpoints that are represented as CORBA objects and that are managed outside the scope of the ODBMS. Hence, an integration between ODBMSs and CORBA is required to facilitate integrated viewpoint based environments.

In order to enable an external tool to invoke an operation from an object stored in an ODBMS the operation has to be specified in IDL and its implementation has to invoke the operation of the ODBMS schema. Some ODBMSs, such as O_2 and ObjectStore provide facilities to generate IDL interfaces and their implementation directly from the schema. If these generation facilities are not provided, the IDL interfaces have to be hand-coded. Any ODMG-93 compliant ODBMS has a C, C++ or Smalltalk language binding to implement the methods of the schema. For any of these languages IDL language bindings that determine how IDL operations are implemented are also standardised. Finally, the skeleton and the method implementation have to be registered in the CORBA implementation repository so that they are activated as soon as the external tool issues the operation execution request.

Attributes that implement inter-subgraph semantic edges, i.e. edges that

lead to subgraphs that are potentially managed outside the ODBMS, are implemented in the ODBMS schema as attributes that store externalised CORBA object references as character strings. If the ODBMS is ODMG-93 compliant, the schema will be implemented in C, C++ or Smalltalk. Using the respective CORBA IDL binding to C, C++ or Smalltalk, methods of the ODBMS schema can internalise a CORBA object reference and then use the CORBA static or dynamic invocation interface to invoke an operation belonging to a subgraph that is managed by a remote CORBA object server.

An ODBMS transaction can guarantee integrity and serialisability of operations that modify objects managed under control of the ODBMS. ODBMS transactions, however, cannot guarantee integrity and serialisability of accesses to objects managed outside the scope of the ODBMS. This is the case if builders of viewpoint-based tools have decided not to implement subgraphs in an ODBMS schema, or to store them in different databases managed by different servers. Then distributed transactions have to be employed and the different ODBMS transaction managers have to participate. The CORBA transaction service supports these distributed transactions by using a two-phase commit protocol.

Most database systems, be they relational or object-oriented, support the XA transaction protocol defined by the X/Open Group. The XA protocol standardises an application programming interface that is used during two-phase commit. Using the XA interface, the CORBA transaction service can be implemented in a way such that databases participate in distributed transactions. The CORBA transaction service defines interfaces for transactional clients, transactional servers and transaction coordinators. A tool issuing a command that involves more than one abstract syntax graph database server would be a transactional client and request the transaction coordinator to begin a transaction. Any object database participating in that distributed transaction registers itself with the implementation of the transaction coordinator using the XA protocol. When the tool wants to complete the tool command it requests a commit from the coordinator. To implement that commit the coordinator uses (in the first phase) the XA protocol to obtain completion votes from any participating database. If all participating databases express that they are able to commit, the coordinator implementation would use the XA interface to ask all databases to commit (in the second phase). The implementation of transactional servers, which is very complicated in general due to server uncertainty and a necessity of forward recovery, is fully achieved by the participating databases as part of the XA protocol implementation.

3.5 Summary

The architecture that has been developed in this section is summarised in Figure 3. At a conceptual level, different viewpoint representations form one project-wide abstract syntax graph. Edges between nodes of subgraphs representing different viewpoints are used for checking and preserving inter-viewpoint consistency.

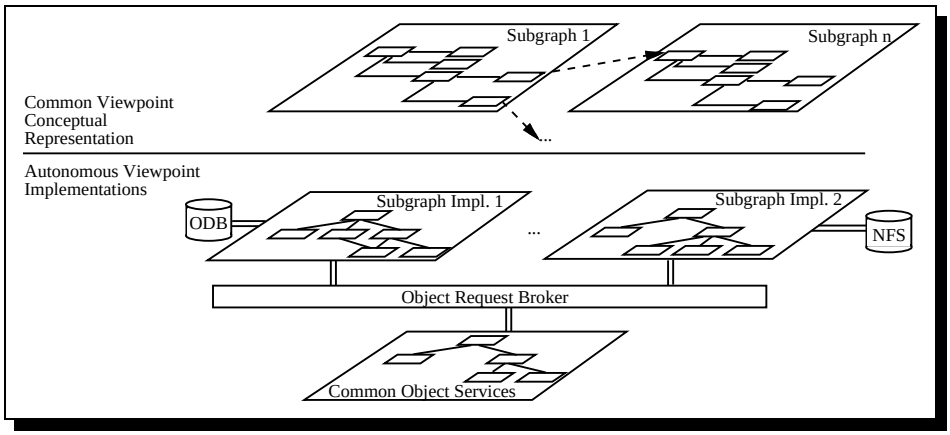

Figure 3: Architecture for Viewpoint-based Environments

The different subgraphs, however, can be implemented in a fairly autonomous manner. Interoperability between the distributed and heterogeneous viewpoint representations is achieved using CORBA. An object request broker may integrate tools, for instance constructed using object databases running on a UNIX platform, with tools that might store viewpoint representations in the file system and operate on a Windows platform. The broker resolves differences in data representations and programming languages. The CORBA transaction service is used to implement transactions that span the boundaries of single database.

4 Related Work

The idea of using syntactic structures for viewpoint representation was developed during the early eighties in a number of projects including Gandalf [11] and the Cornell Synthesizer Generator [15]. Tools generated by these systems are used in programming environments, that means they are intended to support viewpoints in one language only. Therefore, they initially did not address the problem of inter-viewpoint consistency constraints. Garlan suggested the use of different views of the same common representation [10]. As there is only one common conceptual representation this approach, however, removes the possibility of inter-viewpoint consistency constraint violations. Moreover, none of these early tools provide sufficient support for concurrency control; most store viewpoints in a flattened representation in the file system. In the circumstance of concurrent tool execution, this implies that changes done by one tool to a viewpoint are lost as soon as some other tool, which had read the viewpoint before the other has written it, writes its changes to the file system. Version management is also not supported explicitly by any of these tools.

A proof-of-concept prototype for viewpoints was provided in the Viewer en-

vironment [14]. This prototype was implemented in Smalltalk. While demonstrating the appropriateness of viewpoint concepts, the Viewer prototype does not address version management, distribution or heterogeneity, as our architecture does. Tools within the viewer run in the same process and concurrent tool execution is not addressed. Persistence is achieved by storing an image of that process and transaction management that is needed for supporting multiple users is not addressed at all.

The IPSEN environment [13] was among the first environments that considered inter-viewpoint consistency. Tools checking these inter-viewpoint consistency constraints are physically executed in one process. Hence, the basic mechanism to achieve inter-viewpoint consistency is implemented by procedure calls and the integration of autonomously constructed tools into the IPSEN environment is not addressed. Cooperative support is not supported since the home grown database system underlying IPSEN has only very limited concurrency control facilities.

5 Summary and Further Work

As our motivation, we have summarised how viewpoints can be used as a conceptual framework for the integration of software development methods and tools. We have discussed project-wide abstract syntax graphs for viewpoint representation and persistent storage of viewpoints. We have then outlined how syntax graphs can be implemented as distributed CORBA objects. For tools that can be constructed anew we have indicated how persistence, version management and transaction support can be achieved using ODMG object databases. Finally, we have sketched how CORBA and ODMG databases can be combined to achieve interoperability of heterogeneous and distributed viewpoint based tools.

The construction of tools on top of object databases has been fully validated and explored [6]. We have preliminary but promising experience with a-priori, CORBA-based integration of tools using different object database servers (of the same type).

The approach outlined in this paper, however, will only achieve its full potential if we can show that it is generally applicable to the integration of off-the-shelf software development tools. In order to make such a general claim, we will have to demonstrate that every off-the-shelf tool can be wrapped with one or more CORBA objects. To support this claim a more detailed survey of the integration mechanisms that are available in current off-the-shelf tools will have to be made.

We believe that our solution to integration of viewpoint development tools is also applicable to other information systems, where information is kept in different autonomous and heterogeneous data stores. We have yet to investigate this possibility.

Acknowledgements

My colleagues Stephen J. Morris and George Spanoudakis provided me with valuable comments on earlier drafts, which allowed me to improve contents and presentation of the paper.

References

[1] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System: the Story of O_2*. Morgan Kaufmann, 1992.

[2] N. S. Barghouti, W. Emmerich, W. Schäfer, and A. H. Skarra. Information Management in Process-Centered Software Engineering Enviroments. In A. Fuggetta and A. Wolf, editors, *Software Process*, number 4 in Trends in Software, chapter 3, pages 53–87. Wiley, 1996.

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

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

[5] W. Emmerich. Tool Specification with GTSL. In *Proc. of the 8^{th} Int. Workshop on Software Specification and Design, Schloss Velen, Germany*, pages 26–35. IEEE Computer Society Press, 1996.

[6] W. Emmerich, J. Arlow, J. Madec, and M. Phoenix. Tool Construction for the British Airways SEE with the O_2 ODBMS. *Theory and Practice of Object Systems*, 1997. To appear.

[7] W. Emmerich, P. Kroha, and W. Schäfer. Object-oriented Database Management Systems for Construction of CASE Environments. In V. Mařik, J. Lažanksý, and R. R. Wagner, editors, *Database and Expert Systems Applications — Proc. of the 4^{th} Int. Conf. DEXA '93, Prague, Czech Republic*, volume 720 of *Lecture Notes in Computer Science*, pages 631–642. Springer, 1993.

[8] 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.

[9] A. Finkelstein, D. Gabbay, H. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency Handling in Multi-Perspective Specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.

[10] D. Garlan. *Views for Tools in Integrated Environments*. PhD thesis, Carnegie Mellon University, 1987.

[11] A. N. Habermann and D. Notkin. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, 1986.

[12] ISO 8879. Information processing – Text and Office Systems – Standardised General Markup Language SGML. Technical report, International Standards Organisation, 1986.

[13] M. Nagl. An Incremental and Integrated Software Development Environment. *Computer Physics Communications*, 38:245–276, 1985.

[14] B. Nuseibeh, J. Kramer, and A. Finkelstein. A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, 1994.

[15] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator – a system for constructing language based editors.* Springer, 1988.

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

[17] L. Vidal, A. Finkelstein, G. Spanoudakis, and A. L. Wolf, editors. *Viewpoint '96. In Joint Proc. of the SIGSOFT '96 Workshops.* ACM Press, 1996.