Developing Collaborative XML Editing Systems

Ansgar Robert Sandy Gerlicher

A thesis submitted in fulfilment of the requirements of the University of the Arts London for the degree of Doctor of Philosophy

October 2007

London College of Communication

University of the Arts London

Abstract

In many areas the eXtensible Mark-up Language (XML) is becoming the standard exchange and data format. More and more applications not only support XML as an exchange format but also use it as their data model or default file format for graphic, text and database (such as spreadsheet) applications.

Computer Supported Cooperative Work is an interdisciplinary field of research dealing with group work, cooperation and their supporting information and communication technologies. One part of it is Real-Time Collaborative Editing, which investigates the design of systems which allow several persons to work simultaneously in real-time on the same document, without the risk of inconsistencies.

Existing collaborative editing research applications specialize in one or at best, only a small number of document types; for example graphic, text or spreadsheet documents. This research investigates the development of a software framework which allows collaborative editing of any XML document type in real-time. This presents a more versatile solution to the problems of real-time collaborative editing.

This research contributes a new software framework model which will assist software engineers in the development of new collaborative XML editing applications. The devised framework is flexible in the sense that it is easily adaptable to different workflow requirements covering concurrency control, awareness mechanisms and optional locking of document parts. Additionally this thesis contributes a new framework integration strategy that enables enhancements of existing single-user editing applications with real-time collaborative editing features without changing their source code.

Publications and presentations based on this research thesis

Publications

- *Compendium of Computer Science in Media (Kompendium der Medieninformatik – Medienpraxis).* Authored chapter on Computer Supported Cooperative Work (CSCW). Springer Verlag. July 2007, 315 pages. ISBN 978-3-540-36629-4
- *Transparent Extension of Single-User Applications to Multi-User Real-Time Collaborative Systems - An Aspect Oriented Approach to Framework Integration.* Proceedings of the 9th International Conference on Enterprise Information Systems – ICEIS 2007. Funchal, Portugal 2007, pp. 327-334. ISBN 978-972-8865-91-7
- *A Framework for Real-Time Collaborative Engineering in the Automotive Industries.* Lecture Notes in Computer Science (LNCS 4101). Proceedings of the 3. International Conference on Collaborative *Design*, Visualization and Engineering. Mallorca, Spain. Springer Verlag 2006, pp. 164-173, ISBN 3-540-44494-7
- *Extending existing single-user applications with collaborative functionality using the Collaborative Editing Framework for XML (CEFX). (Erweiterung bestehender Anwendungen um kollaborative Funktionen mit Hilfe des Collaborative Editing Framework for XML (CEFX)).* In Proceedings of the doIT Software-Research-Day (Software-Forschungstag). Stuttgart. Oktober 2004. Fraunhofer IRB Verlag 2005, pp. 150-165, ISBN 3-8167-6715-X

Presentations

- 9th *International* Conference on Enterprise Information Systems, Funchal, Madeira, Portugal (ICEIS 2007, 06/2007)
  Title of the talk: *Transparent Extension of Single-User Applications to Multi-User Real-Time Collaborative Systems - An Aspect Oriented Approach to Framework Integration*
- 3. Automotive Harness Forum, Munich, Germany (10/2006)
  Title of the talk: *Collaborative Engineering – Die Bordnetzenwicklung der Zukunft (the future of vehicle electrical engineering)*
- Third International Conference on Cooperative Design, Visualization and Engineering, Mallorca, Spain (CDVE2006, 09/2006)
  Title of the talk: *A Framework for Real-Time Collaborative Engineering in the Automotive Industries*
- Spring Research Symposium, RNUAL, London, UK (02/2005)
  Title of the talk: *Developing a method for web-based collaborative editing of XML documents which will improve collaborative editing procedures*
- Do IT Software-Research-Day, Stuttgart, Germany (10/2004)
  Title of the talk: *Collaborative Editing Framework for XML Erweiterung bestehender Anwendungen um kollaborative Funktionen (extending existing single-user applications with collaborative functionality using)*
- European Computer-Supported Cooperative Work Conference 2003, Doctoral Colloquium, Helsinki, Finland (ECSCW03, 09/2003)
  Title of the talk: *Real-time web-based collaborative editing of XML documents - editing structured documents*

Acknowledgements

I wish to express my most sincere gratitude to my director of studies Dr Jack Tchan and my supervisors Professor Robert Thompson and Professor Martin Goik for their great supervision and guidance during this study. I also would like to sincerely thank Dr. Andrew Manning as part of the "London team" for his support as a substitute supervisor. I was always cordially received in London and it has been a great pleasure working with them. I would also like to express my gratitude to all members of staff at the LCC that I made contact with. Thanks also goes to Professor Wolfgang Faigle. Without his support this would not have been possible. Not to forget my colleagues and Professors at the HdM Stuttgart, where I spent three great years of my work life. They gave me all the freedom and support I needed in order to get this research started. I also would like to thank Dr. Claudia Ignat and Professor Hala Skaf-Molli for their support and for providing me with valuable information, at the very beginning of my studies, which helped me to orientate and better understand the research field of real-time collaborative editing. Special thanks goes to my lovely grandaunt Vera who always gave me a home during my visits in London. The same applies to my grandmother Jean and the rest of the British family. Thanks also goes to my parents for their support and for believing in me. Last but not least, I would like to thank Kathrin for her patience, comprehension and emotional support especially at those many weekends I had to work on the thesis.

Table of Contents

List of Figures

Chapter 1. Introduction

With the advent and popular use of computer network systems has arisen the concept of collaborative editing of computer files. These computer files can contain either image data used in arts and multimedia applications or text data used in publishing, as well as data from many other domains. Contemporary real-time collaborative editing systems can be used for one type of data, either image or text only; for example GRACE (Sun and Chen 2002), REDUCE (Chen 2001). Editing of image data and text data in one system is not possible.

The Extensible Mark-up Language (XML) is a de facto standard today. Because of its wide spread use as a storage format for image data as well as for text data it is very important for publishing, data exchange, graphics and many other application areas. It can be regarded as a unified representation of many of the computer file representations. In this thesis a system for collaborative editing of XML files is devised which unifies different collaborative editing solutions for different application types. This is achieved by the development of a framework for concurrent editing of XML documents. The framework provides a programmers interface for developing new real-time collaborative editing applications or augmenting existing single-user XML editing applications with collaborative editing features.

> "Synchronous collaborative editing systems need concurrency control schemes to resolve inconsistency problems caused by participants' simultaneous operations." (Chen 2001).

This thesis introduces a concurrency control scheme for synchronous collaborative editing of XML documents. It examines to what extent the characteristics of XML can support synchronous collaborative editing. A collaborative editing system is used by team members who work together to achieve a common goal. Teams sometimes consist of globally distributed members. Therefore a model for a distributed collaborative editing system for XML documents is introduced herewith. To understand how a collaborative editing system for XML documents differs from other collaborative editing systems it is important to know the properties of XML and what makes XML different to other computer file formats. The following sections 1.1 to 1.4 give a short introduction to the emergence of XML, the difference between programming

languages and XML languages and the common structure of XML documents. Section 1.5 discusses concurrency problems and concurrency control techniques in database and collaborative systems. Systems architectures in distributed systems are discussed in section 1.6. Section 1.7 discusses contemporary web-based editing systems and the Hypertext Transfer Protocol (HTTP). The last section of the first chapter outlines this thesis and its contributions to the field of Computer Supported Cooperative Work (CSCW).

1.1. Emergence of XML

When Tim Berners-Lee et al invented the Hypertext Markup Language (HTML) in 1989 at CERN[1] they wanted to have a simple method for hypertext based information exchange, that would easily work on different computer systems in use at CERN. At that time many people were using TeX[2] and PostScript[3] for their scientific documents. A few were using the Standard Generalized Markup Language (SGML)[4]. Something simpler was needed that could be used with any type of terminal or workstation running X Window[5] (MIT-X11). HTML - a SGML application - was designed for one purpose: the presentation of information on a TCP/IP based network - the nucleus of today's Internet. In 1993 it was published as an Internet Draft. SGML has a more general usage. As a so called Meta-language it is used to define Mark-up languages such as HTML. But SGML was too complicated to be successful and thus XML, a subset of SGML was defined in 1998. XML - a much simpler but as flexible language as SGML - was originally (like its predecessor) designed to meet the challenges of large-scale electronic publishing. Today XML also plays an increasingly important role as an exchange format between applications[6]. XML became very successful because of its simplicity. It is relatively human-legible and at the

---

1 European Organization for Nuclear Research, CERN, Genève, Switzerland

2 TeX typesetting system invented by Donald Knuth

3 PostScript programming language for printing graphics and text. Introduced by Adobe in 1985.

4 SGML: ISO 8879:1986 Information Processing — Text and Office Systems

5 The X Window System provides a way of writing device independent graphical and windowing software that can be easily ported from machine to machine. It is maintained by X.Org. Retrieved July 15, 2007 from http://www.x.org

6 The role of XML today: http://www.w3c.org/XML/, retrieved October 30, 2007

same time it facilitates processing by a computer. Today a vast number of applications exist that are defined in XML. For example, the successor of HTML, called XHTML[7] is defined as an XML application. Other examples are X3D[8] (an XML compliant successor of VRML[9]), SVG[10] and MathML[11].

1.2. Meta-syntax for the specification of programming languages

Every computer language has a grammar that defines it. Almost every programming language can be described using the Backus Naur (Nauer et al. 1963) Form (BNF). BNF[12] is a so called formal meta-syntax to express context-free grammars. It can be used to describe the syntax and the formal grammar (context –free grammar[13]) of a programming language. For special cases, where BNF or EBNF (Enhanced Backus Naur Form) cannot be used, the rules are defined in a descriptive way. Once a programming language has been defined, it can be used to write a program.

1.2.1. Parsing of programs

A parser is a computer program or a component of a program that analyses a sequence of tokens and determines the grammatical structure of an input with respect to a given formal grammar. Before a parser can start to work, a lexical analysis of the program's source code is performed by a so called scanner. The scanner checks the input for lexical correctness and generates a token sequence, which is passed to the parser (see figure 1.1).

---

7 Extensible Hypertext Markup Language: http://www.w3c.org/MarkUp/, retrieved October 30, 2007

8 EXtensible 3D. Draft specification committed to ISO/IEC JTC1/SC24 for registration December 2002: http://www.web3d.org/x3d.html, retrieved October 30, 2007

9 Virtual Reality Modelling Language. International Standard ISO/ IEC 14772-1,1997

10 Scalable Vector Graphics 1.0 Specification , 2001: http://www.w3.org/TR/SVG/, retrieved October 30, 2007

11 Mathematical Mark-up Language (MathML) 2.0. , 2001: http://www.w3.org/TR/MathMl2/, retrieved October 30, 2007

12 Backus Naur Form: a formal notation to describe the syntax of a given language. John Backus and Peter Naur in 1960

13 From Wikipedia (http://www.wikipedia.org), retrieved October 30, 2007: A formal language is context-free if there is a context-free grammar which generates it

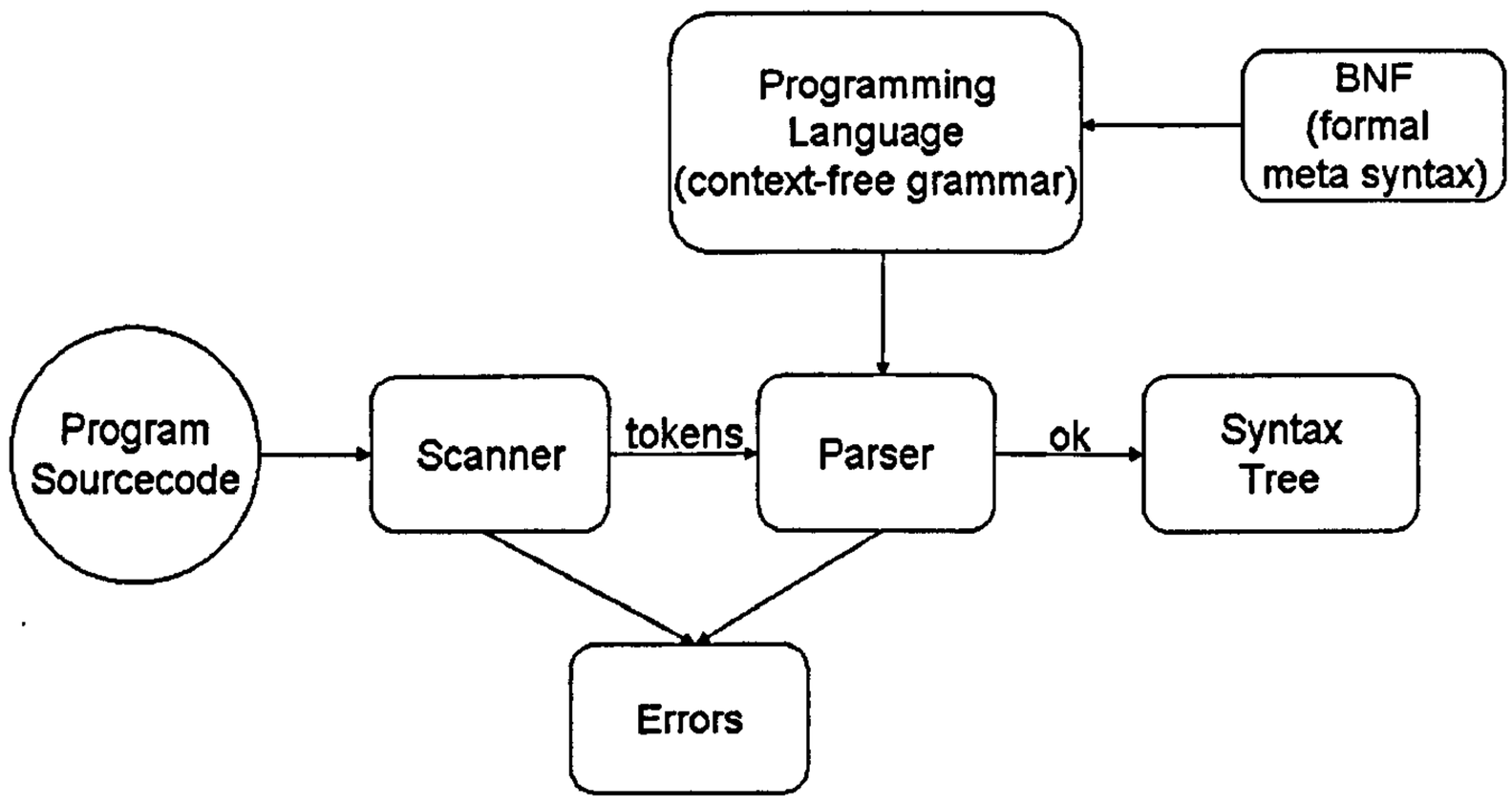

Figure 1.1: Parsing of a computer program

The job of the parser is to check if the token sequence provided by the scanner, complies to the grammar of the respective programming language. If no lexical or grammatical errors are detected the parser builds a data structure such as a concrete or abstract syntax tree. The syntax tree is then further processed by a semantic analyser.

"hello"	Lexically correct
"xjsk§?"	Lexically incorrect

Table 1.1: Example for lexically correctness in the English Language

In the English language, for example, a token sequence – that is a sequence of words - can form a sentence if the token sequence follows certain grammatical rules.

“Paul is no chair hello”	Lexically correct but grammatically incorrect
“Paul sits green fish”	Grammatically correct but semantically incorrect

Table 1.2: Example for grammatical correctness in the English Language

1.2.2. Semantic correctness

As seen above, the sentence “Paul sits green fish” is grammatically correct but semantically incorrect. It just does not make sense. To recognise the meaning of a

sentence, the words have to be seen in their context. Similar problems occur in computer languages. Parsers for programming languages that are defined using a context-free grammar do not check the semantic correctness of a program's source code. The semantic of a program is checked by a semantic analyser.

`int x = "no int"++`	Syntactically correct but semantically incorrect because the types do not match. The statement "no int" is not an integer and can not be incremented.
`int y = 1 + 2;`	Semantically correct, 1 and 2 are of the type integer.

Table 1.3: Example for semantically incorrect program code

Semantic analysis is the process of examining the types and values of the statements used to make sure they make sense. During the semantic analysis, the types, values, and other required information about statements are recorded, checked and transformed to enable the code generator to produce unambiguous executable machine code or intermediate code (e.g. in the case of the Java programming language). Table 1.3 shows an example for semantically incorrect and correct C/C++ program code.

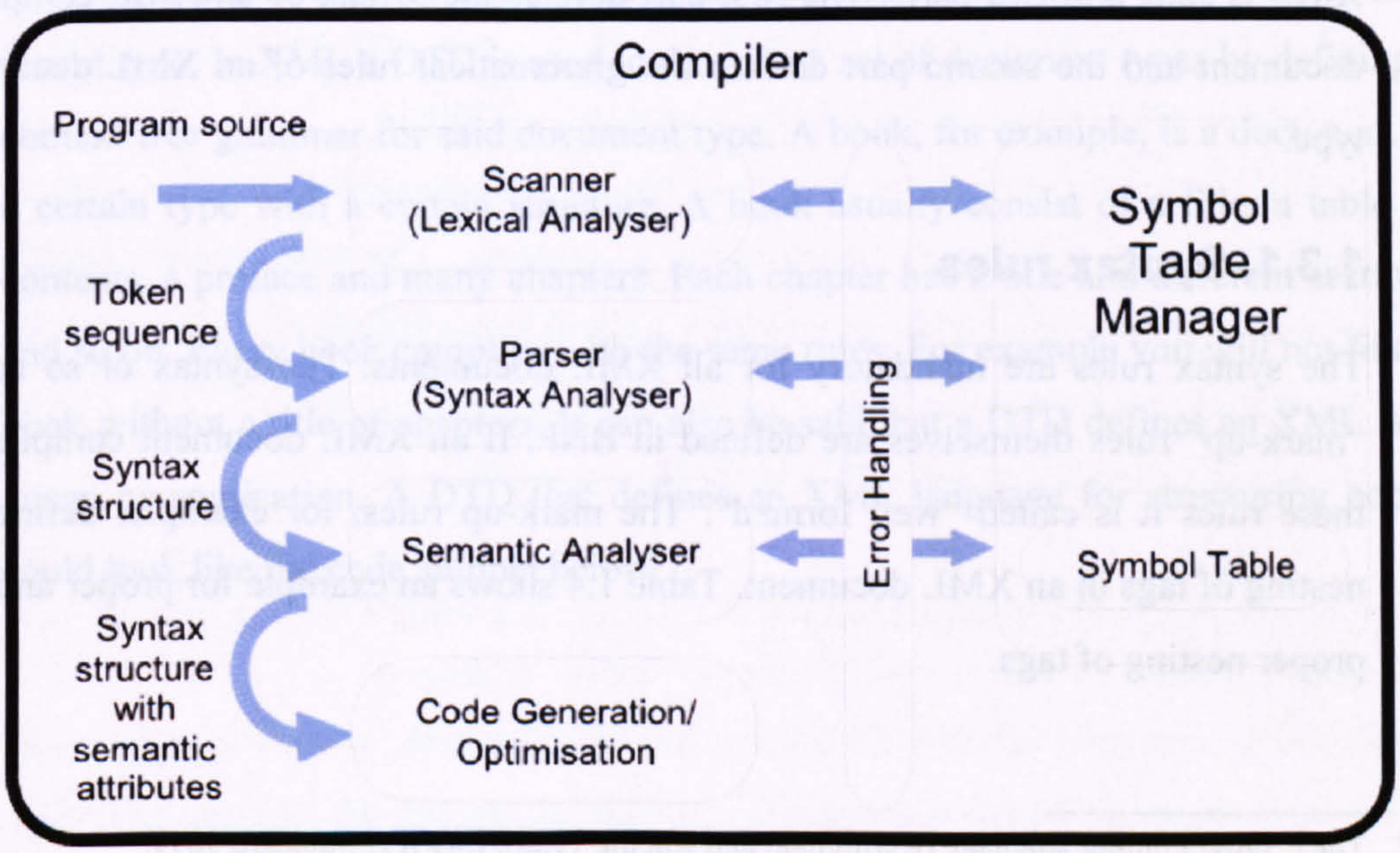

Figure 1.2: Compiler tasks and structure

A computer program language is an instruction language that uses instructions and commands and defines actions that are then carried out by an executive unit. Computer programs usually are parsed, semantically analysed and then translated (code generation and optimisation) into a machine language or an intermediate language. This process of scanning, parsing, semantic analysis and translation is performed by a computer program called a compiler (figure 1.2). The executive unit in that case can be the core processing unit (CPU) of a computer (compiled code) or an interpreter (in the case of an interpreted programming language).

This characterises the main difference of programming languages to XML: In general XML is not an instructing language. XML is a language to structure information and is in most cases only parsed and not compiled or interpreted. Thus we will restrict our discussion to parsing of XML documents in this introduction.

1.3. XML for the specification of XML languages

XML is a de facto standard specified by the Word Wide Web Consortium (W3C). In contrast to SGML it is not an ISO (International Organisation for Standardisation) standard. As mentioned, XML is not a programming language like for example C[14] or Pascal[15] but a language to structure information. It is a so called Meta Language, but there are still many similarities to programming languages. The definition of XML is split into two parts. The first part defines the syntax of an XML compliant document and the second part defines the grammatical rules of an XML document type.

1.3.1. Syntax rules

The syntax rules are mandatory for all XML documents. The syntax or so called "mark-up" rules themselves are defined in BNF. If an XML document complies to these rules it is called "well formed". The mark-up rules, for example, define the nesting of tags in an XML document. Table 1.4 shows an example for proper and improper nesting of tags.

---

[14] The C programming language (Kernigham and Ritchie 1988), ISO/IEC Standard 9899.

[15] The Pascal programming language was developed by Niklaus Wirth in 1971, ISO Standard 7185.

<book> <title>MyBook</title> <chapter> <heading>XML</heading> </chapter> </book>	Proper nesting of tags
<book> <title>MyBook</title> <chapter> <heading>XML</heading> </book> </chapter>	Improper nesting of tags

Table 1.4: Example for proper nesting and improper nesting

1.3.2. Grammatical rules

1.3.2.1. DTD

The grammatical rules of an XML document type can be defined in a so called Document Type Definition (DTD). The DTD is based on a meta-syntax that itself is defined in BNF. An XML document type is a document of a specific type. For example a book is one document type and an application form is another document type. Each document type has certain properties that distinguish it from another document type. In XML a DTD is used to describe a set of document types by defining a context free grammar for said document type. A book, for example, is a document of a certain type with a certain structure. A book usually consist of a title, a table of contents, a preface and many chapters. Each chapter has a title and different sections and so on. Every book complies with the same rules. For example you will not find a book without a title or chapters. It can also be said that a DTD defines an XML language or application. A DTD that defines an XML language for structuring books could look like the code snippet below.

```
<!ELEMENT book (title, subtitle*, bookinfo?, toc?,
      (dedication | preface)*, (chapter | part)*,
      (appendix | bibliography | colophon | glossary |
      reference)*) >
<!ELEMENT chapter (indexterm*, title, chapterinfo?,
      (indexterm | refentry | simplesect | sect1 |    section )*) >
```

This excerpt from the DocBook Lite[16] DTD is an example for an XML language definition. For a DTD the text content of a document it is not relevant, but the structure is. It would not be relevant, for example, what the title of a single book is but it is relevant that the title is mandatory. It can be said that in XML, document types are specifications for content types and their relations (Michel 1999). That is a DTD defines which elements are allowed in a specific XML document type and how these elements can be mixed to obtain a valid document instance. The XML 1.0 specification provides the syntax of DTDs and the rules for the XML parsers to validate the documents on the basis of the DTD (Arciniegas 2002). As mentioned before, a DTD specifies hierarchy, containment and cardinality of elements for a document type. The term hierarchy relates to the order of elements in a document. The term containment relates to the content type of possible elements, for example, text, numbers, character entities or other elements. Referring to cardinality, a DTD can specify that, for example, an element can occur zero times or once, zero times or arbitrary times or once or arbitrary times within an element in a document.

A DTD has some limitations. For example it is not possible to specify, that an element can occur only twenty times within a document. The most serious limitation is probably the missing typing of character data. It is not possible to determine that the content of an element has to be a certain type of string or number. This can lead to misuse of certain elements within an XML document:

```
<phone>0711-685-8362</phone>
<phone>this is not a valid phone number</phone>
```

[16] DocBook Lite. DTD, O'Reilly. V.1.19 , 2003

In the above example this misuse of the phone element is obviously a semantic problem. From the view of the parser, both elements are valid and comply to the element declaration:

```
<!ELEMENT phone (#PCDATA)>
```

These and other limitations of the DTD were the motivation for the development of XML Schema. XML Schemas do a better job of describing data format (hierarchy, containment, and cardinality) and constraints (data type, range and default values).

1.3.2.2. XML Schema

Another way to define the grammar of an XML document type was introduced in May 2001: XML Schema[17]. This W3C recommendation is a de facto standard for defining XML document types. In contrast to DTDs, XML Schema documents are XML documents themselves. This makes it possible to process them with standard XML tools and thus facilitates application's awareness of the underlying grammar. The main advantage of XML Schema in comparison to the DTD is that a document type can be specified in much more detail. XML Schemas have an extensible hierarchy of data types.

An example XML Schema definition:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:complexType name="persondefinition">
  <xs:sequence>
   <xs:element name="firstname" type="xs:string"/>
   <xs:element name="lastname" type="xs:string"/>
   <xs:element name="birthday" type="xs:date"/>
  </xs:sequence>
</xs:complexType>
<xs:element name="person" type="persondefinition"/> </xs:schema>
```

Basic types like string, date and time can be used directly for elements. User defined types are possible. In the above example a data type called "persondefinition" is

[17] http://www.w3.org/XML/Schema, retrieved October 30, 2007

defined. A person has a sequence of the elements "firstname", "lastname" and "birth-day". The element types "xs:string" and "xs:date" are predefined in the XML Schema specification. Besides the definition of data types, one of the features of XML Schema is that it enables the specification of precisely the number of elements that appear within a document. This is done by specifying a "minOccurs" and a "maxOccurs" attribute in the element definition. It is also possible to define keys to unambiguously identify an element within a document. This technique is an extension of the concept of primary keys in relational databases. There are many other advantages of XML Schema over the DTD that in some areas will possibly lead to a substitution of DTDs by XML Schema. One example is named typing which allows to define complex data types. A complex data type is the abstract definition of a structure within an XML document. In the above example "persondefinition" is such a complex data type. This makes it very useful for data exchange (Van der Vlist 2003), (Wyke and Watt 2002), (Walmsley 2001).

1.3.3. Parsing of XML

Similar to computer programs that are written in a programming language, XML documents can be checked for errors. This is done in the same manner as with program source code. An XML document is checked for lexical correctness and tokenized by a scanner. The tokens are the input for the parser. The parser then checks if the document is compliant to the XML mark-up rules. If a document complies to the mark-up rules it is called a "well formed" XML document instance. If the document is "well formed" it can be checked by a parser for compliance to a certain DTD or XML Schema. A parser that is used to check a document on compliance to a Document Type Definition is called a validating parser. A "well formed" XML document that complies to a Document Type Definition, is called a valid XML document.

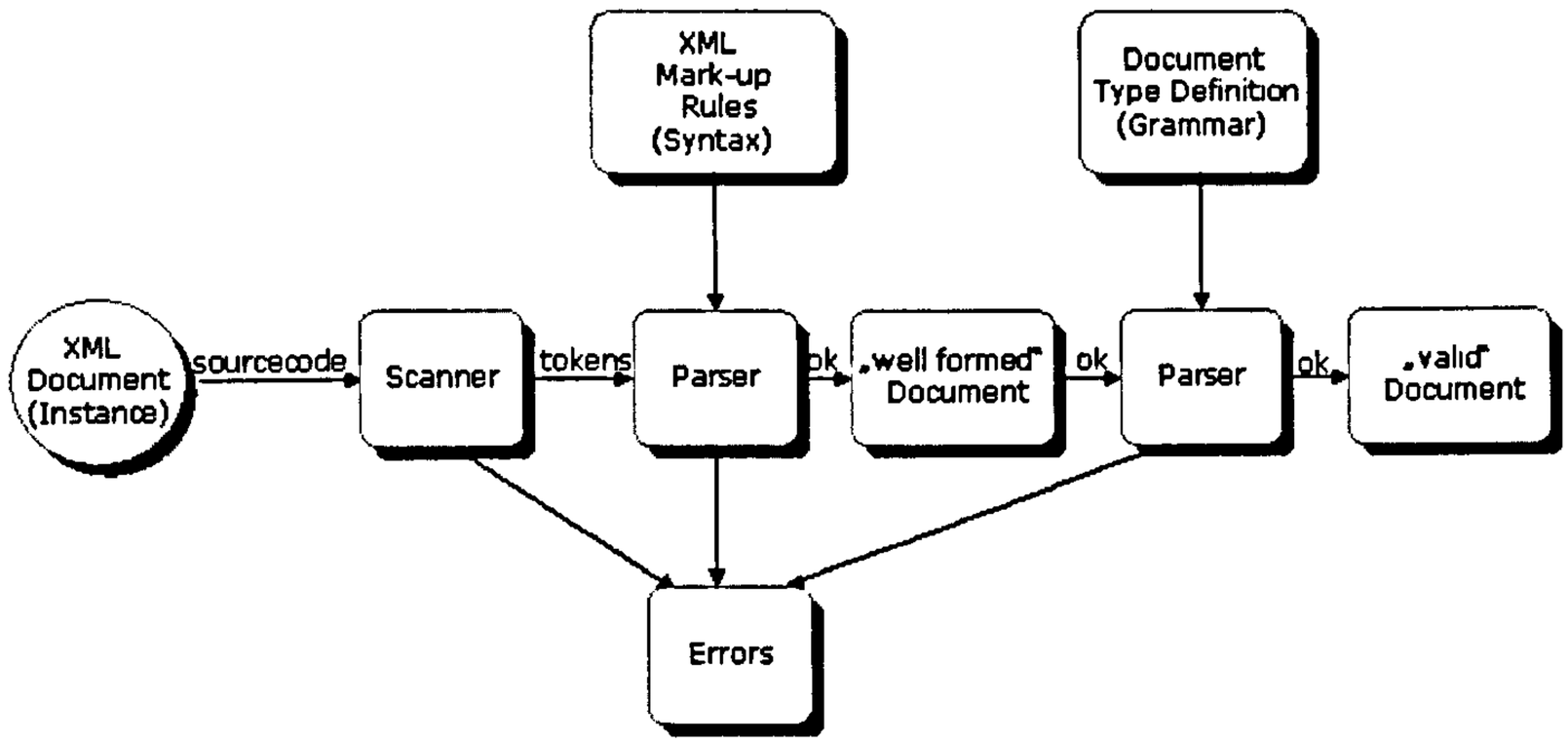

Figure 1.3: Process of XML parsing

If errors occur during the parsing process they are handled accordingly. A document can be "well formed" without specifying a relating DTD. If an XML document does not specify a DTD to which it can be checked on compliance it is not a valid document. In some applications such errors are not relevant and are ignored. In other cases it is critical to the application, that the document is valid and therefore must specify a relating DTD.

1.4. Structure of XML documents

As mentioned earlier, XML is a method to structure information. This structured information is then perceived by a human or processed by a computer. XML documents have a certain type of structure; a hierarchical structure. Thus XML documents can be regarded, in general, as hierarchically structured information. Hierarchically structured information can be represented as ordered trees, in which the children of each node have a designated order. Branch nodes depend on their parent node and have child nodes. Leaf nodes do not have child nodes. Each node in a tree has a label and a value. Figure 1.4 shows a tree representation of an example XML (XHTML) document.

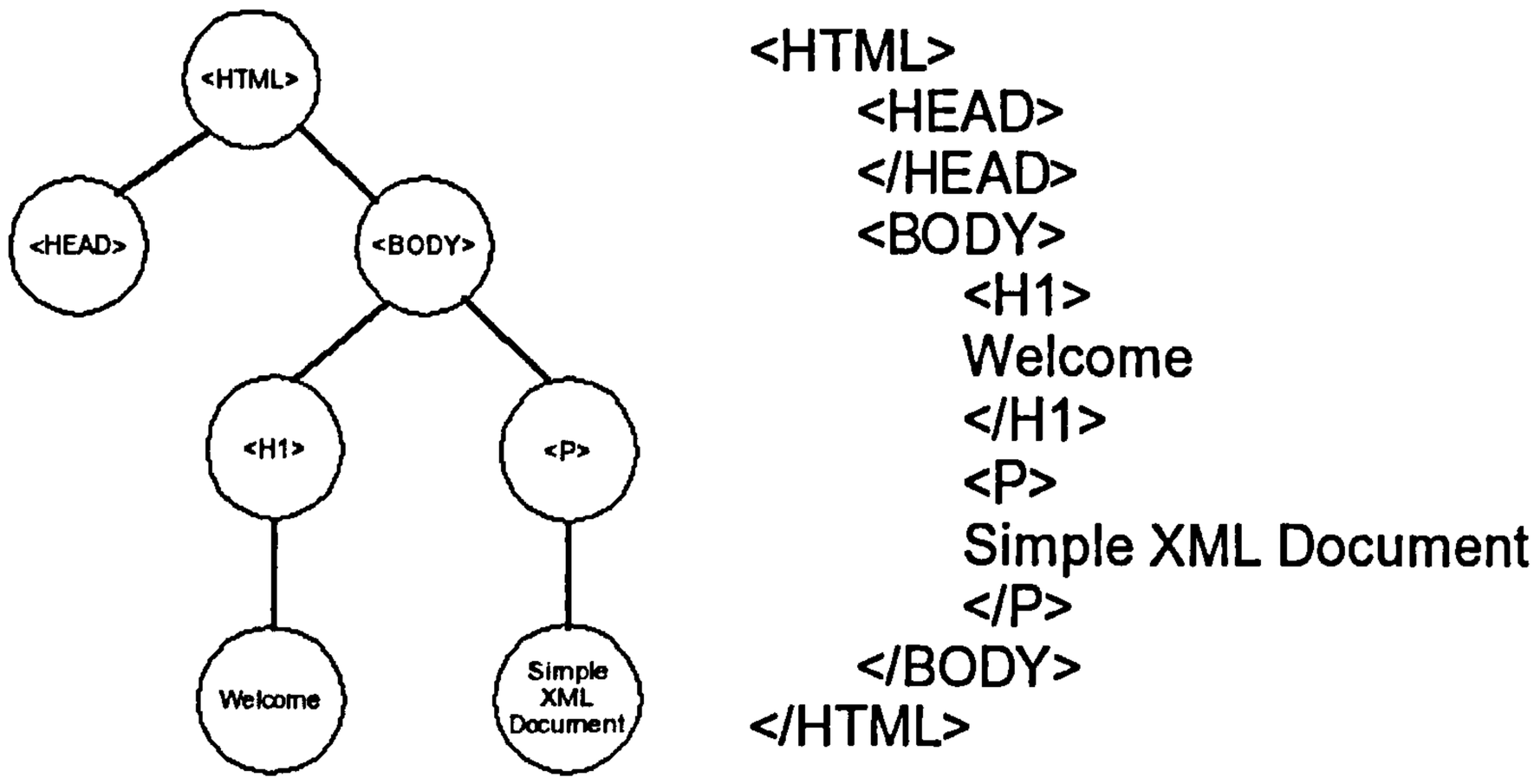

Figure 1.4: Tree representation of a simple XHTML Document

1.4.1. Editing trees

The structure of XML allows us to represent an XML document as an ordered tree, and work with it in that way. Thus editing of XML documents can be generally considered as editing of trees. The editing of trees can be described with four fundamental operations: insert, delete, move and update. The first three are structural operations - they make changes to the structure of the tree. The fourth is a mutational operation - it changes the contents (value) of a node without changing the structure of the tree. In the following these four fundamental operations are briefly discussed.

The *insert* operation creates a node at a given location in the tree. A node x with label l and value v is inserted as the k^{th} child of node y of the tree. The value v is optional, and is assumed to be null if omitted. In order to perform an operation on a node it has to be identified somehow. Different methods for the identification of a node within a hierarchical document exist as is further discussed in chapter 3. In the following examples a node is identified by its name. In figure 1.5 the node *"title"* is inserted as the first child of the node *"head"*.

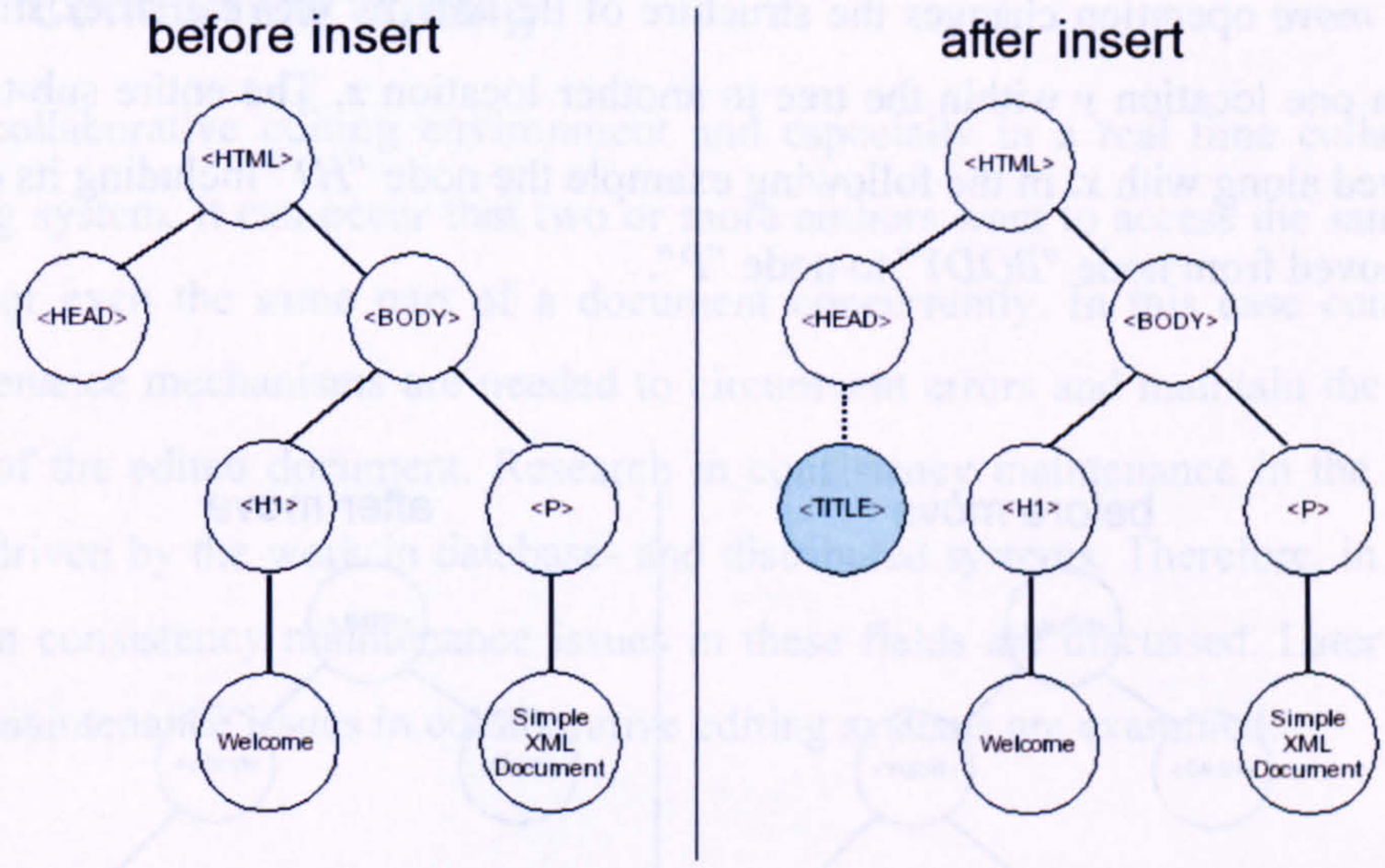

Figure 1.5: Example of an insert operation

The *delete* operation deletes a node *x* from a given location *y* in the tree. This operation only deletes a leaf node; to delete an interior node, first its descendants must be moved to their new locations or deleted. In the following example the child node of the node *"P"* is deleted from the document tree.

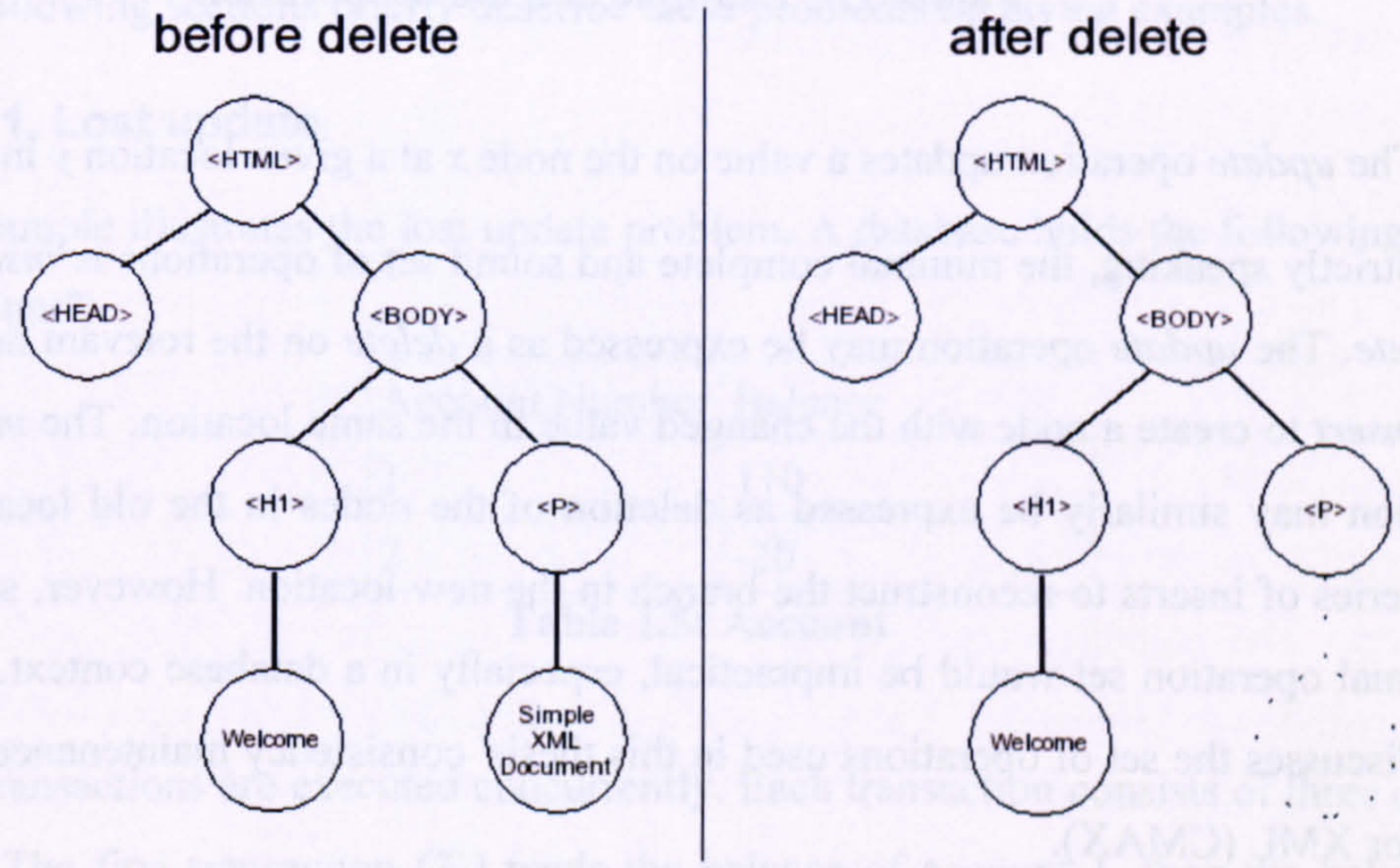

Figure 1.6: Example of a delete operation

The *move* operation changes the structure of the tree by moving an existing node *x* from one location *y* within the tree to another location *z*. The entire sub-tree of *x* is moved along with *x*. In the following example the node *"H1"* including its child node is moved from node *"BODY"* to node *"P"*.

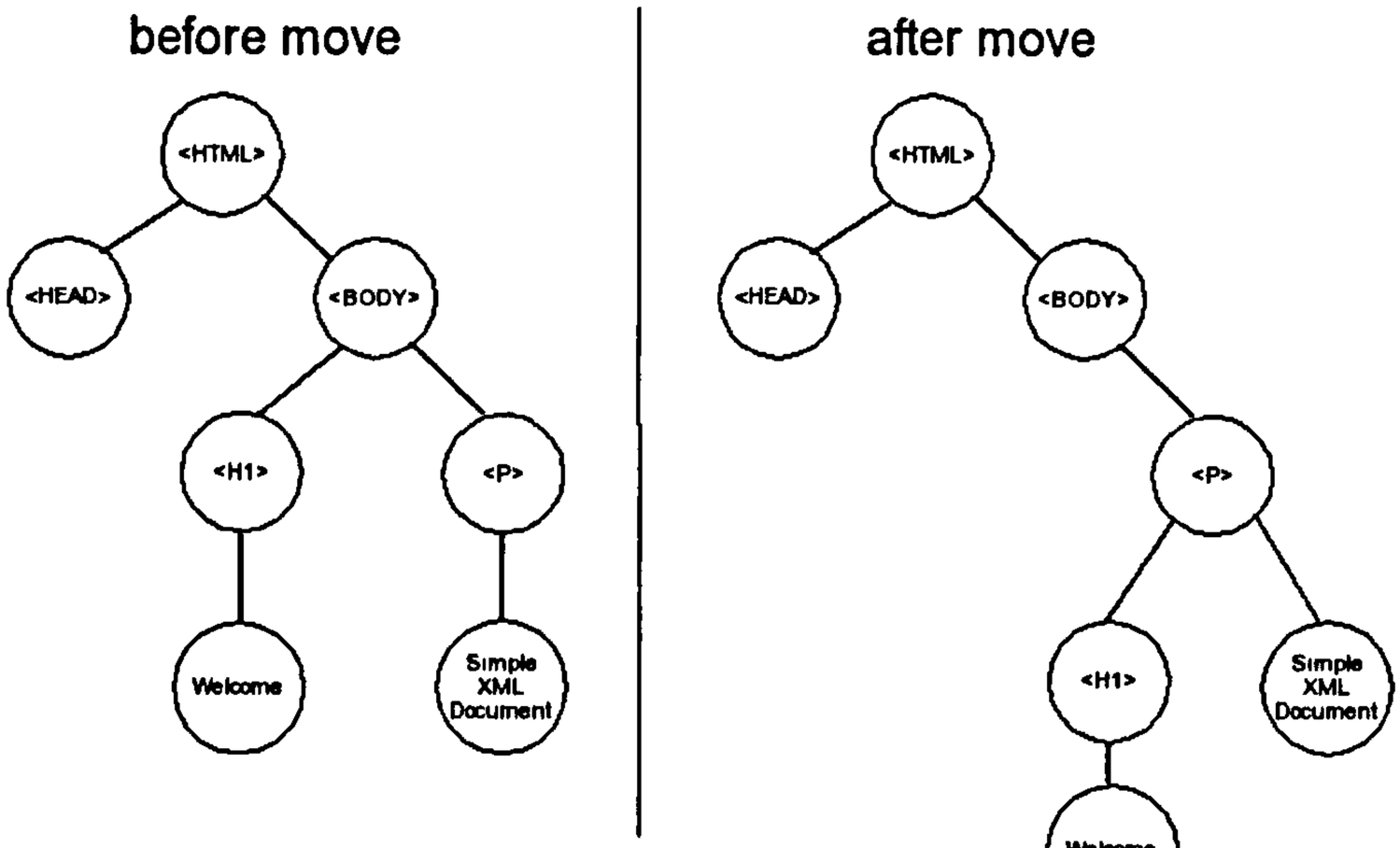

Figure 1.7: Example of a move operation

The *update* operation updates a value on the node *x* at a given location *y* in the tree.

Strictly speaking, the minimal complete and sound set of operations is *insert* and *delete*. The *update* operation may be expressed as a *delete* on the relevant node and an *insert* to create a node with the changed value in the same location. The *move* operation may similarly be expressed as deletion of the nodes in the old location and a series of inserts to reconstruct the branch in the new location. However, such a minimal operation set would be impractical, especially in a database context. Chapter 3 discusses the set of operations used in this thesis' consistency maintenance algorithm for XML (CMAX).

1.5. Concurrent editing

In a collaborative editing environment and especially in a real time collaborative editing system, it can occur that two or more authors want to access the same document or even the same part of a document concurrently. In this case consistency maintenance mechanisms are needed to circumvent errors and maintain the consistency of the edited document. Research in consistency maintenance in the past has been driven by the work in database- and distributed systems. Therefore, in the next section consistency maintenance issues in these fields are discussed. Later consistency maintenance issues in collaborative editing systems are examined.

1.5.1. Concurrency problems in database systems

If several users concurrently access a database, the following problems can occur (Date 2003):

- Lost update
- Uncommitted dependency
- Inconsistent retrieval

The following sections briefly describe these problems by giving examples.

1.5.1.1. Lost update

An example illustrates the lost update problem. A database holds the following table "Account":

Account Number	Balance
1	110
2	-20

Table 1.5: Account

Two transactions are executed concurrently. Each transaction consists of three operations. The first transaction (T_1) reads the balance of account 1, then the balance is decreased in memory by 20. The last step is to write the new value (90) to account 1. The second transaction (T_2) reads the balance of account 1 (110) at nearly the same time as T_1. Then the read value is increased by 30 in memory. In the third step the

new value (140) is written to the database. The result of T_1 is lost because the value written by T_1 (90) is overwritten by the value written by T_2 (140). The execution of the two transaction results in a balance of 140. The final result should be 120, but the update of T_1 was lost.

Time

Transaction 1	Transaction 2	Balance account 1
Read balance from account number 1 into memory		110
Decrease value in memory by 20		110
	Read balance from account number 1 into memory	110
Write new value to balance of account 1 and commit transaction		90
	Increase value in memory by 30	90
	Write new value to balance of account 1 and commit transaction	140

Figure 1.8: Lost update

1.5.1.2. Uncommitted dependency

This is an inconsistency problem which is similar to the lost update problem. For example, this time the balance of account 1 is increased by 30 in the first transaction (T_1) (now it is 140). In the second transaction (T_2) the new value of the account's balance is read into memory (140). Then the first transaction is aborted (because of some error). Now T_2 is based on the wrong value (140 instead 110). The resulting balance value is 120 but it should be 90, because T_1 was aborted (rollback transaction). The reading of the accounts value in T_2 is called a "dirty read". When T_2 modifies the "dirty read" value of account 1 and writes it back to the database this is called a "dirty write".

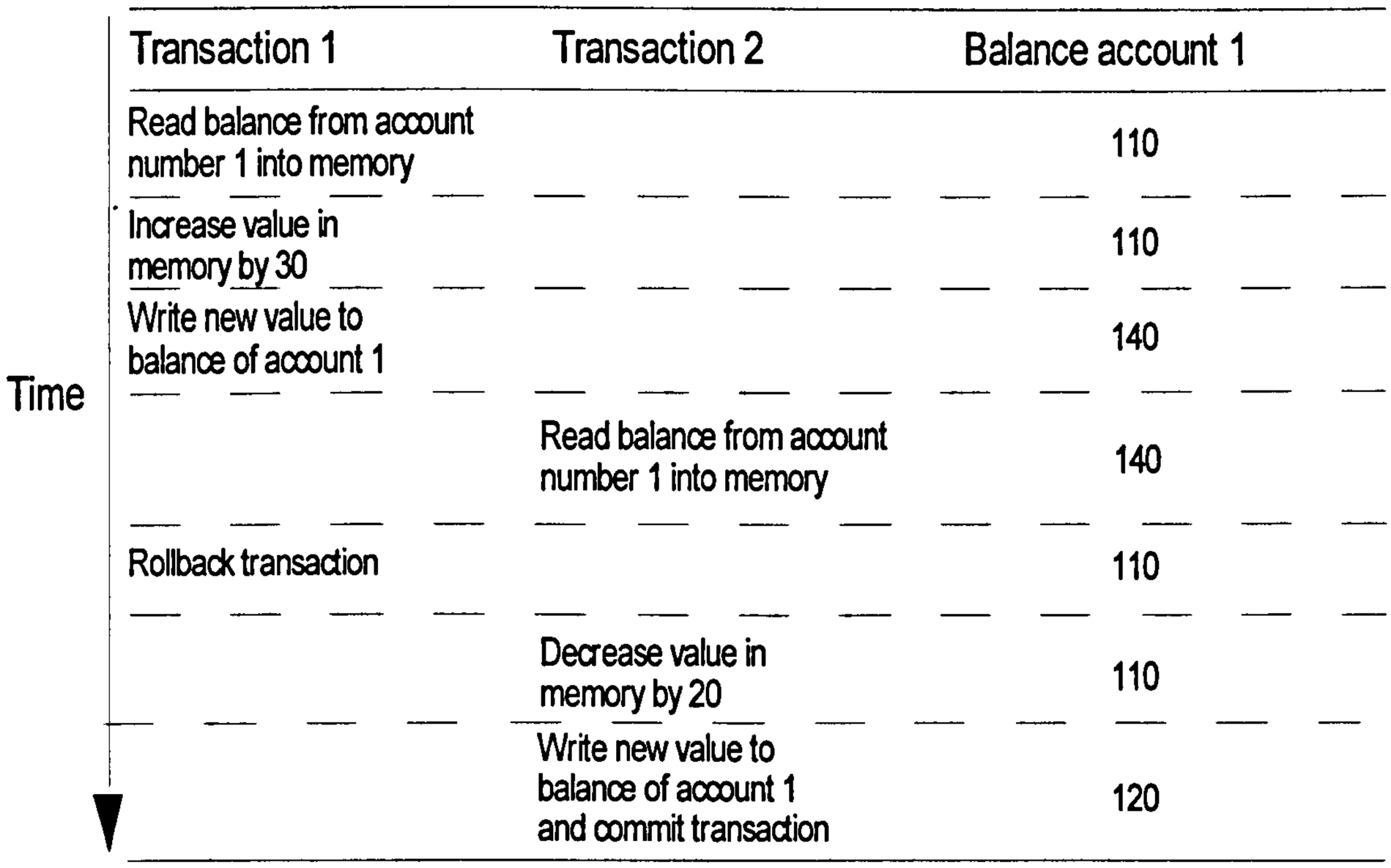

Figure 1.9: Uncommited dependency

1.5.1.3. Inconsistent retrieval

Inconsistent retrieval is another concurrency control problem that occurs when a data item is read in one transaction before another transaction updates it and a second data item is read after being updated by the current transaction. For example, consider a database table like the one in the previous example. Figure 1.10 contains two accounts. A customer transfers money from account 1 to account 2 (transaction T_1). At the same time, another customer starts a second transaction (T_2) and reads the balance of the accounts 1 and 2. The operations of the two transactions (T_1, T_2) are executed according to the schedule shown in figure 1.10.

Time

Transaction 1	Transaction 2	Balance account 1	Balance account 2
Read balance from account number 1 into memory		110	-20
Decrease value account 1 in memory by 30		110	-20
Write new value to balance of account 1		80	-20
	Read balance from account number 1 into memory	80	-20
	Read balance from account number 2 into memory	80	-20
Read balance from account number 2 into memory		80	-20
Increase value account 2 in memory by 30		80	-20
Write new value to balance of account 2		80	10
Commit transaction		80	10
	Commit transaction	80	10

Figure 1.10: Inconsistent retrieval

The first customer transfer interferes with reading the account balance by the second customer. Despite the interference, the values in the database table are still correct but the second customer only sees some of the update transaction results. This means he sees an incorrect account balance of account 2. The problems mentioned above can occur if transactions are interleaved arbitrarily. In the above examples, if T_1 and T_2 were executed one at a time in either order, the results would be correct. If the effect that is produced by the execution of a set of transactions is equivalent to the effect that is produced when these transactions are executed in an arbitrary order, it is called serial equivalence (Bernstein, Goodman et al. 1987). The serial equivalence is one of the consistency properties that have to be preserved by database systems.

1.5.2. Concurrency problems in collaborative editing systems

In collaborative editing systems in general, preserving serial equivalence is very important as well. For example if team members work on a single document concurrently, all copies of the document have to be identical at all sites after executing the same set of operations. To maintain serial equivalence different concurrency

control schemes have been developed, including timestamp ordering and locking (see section 1.5.3). The next sections deal with inconsistency problems in collaborative editing systems that are based on a tree data structure similar to XML tree structures. Examples of such systems are MU3D (Galli and Luo 2000) and GROVE (Ellis and Gibbs 1998). These systems use the *replicated architecture* (see section 1.6.2) to achieve good responsiveness and to avoid a single point of failure in the system. That is, operations that are executed on a local copy of a document are broadcast to the other sites and executed there. Inconsistency problems occur here due to the delay between the execution of an operation at a local site and the execution of the same operation at the remote sites. Figure 1.11 shows an example scenario:

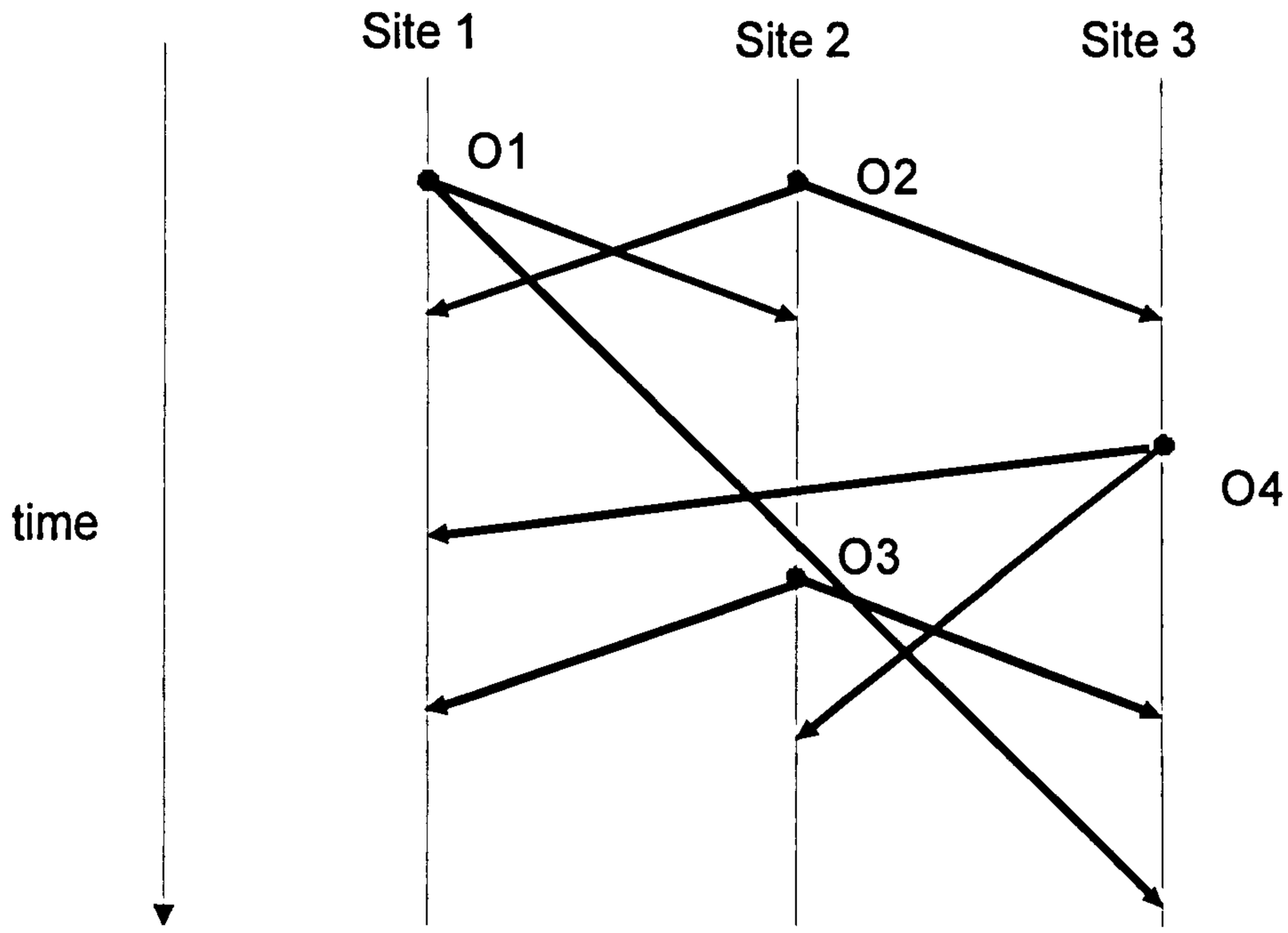

Figure 1.11: Scenario of a real-time group editing session

There are four operations executed O_1,O_2,O_3 and O_4. At site 1 the operations are executed in a different order than at site 2 and site 3. The execution order at site 1 is as follows: O_1,O_2,O_4,O_3. At site 2: O_2,O_1,O_3,O_4 and at site 3: O_2,O_4,O_3,O_1.

1.5.2.1. Divergence

If the operations O_1,O_2,O_4,O_3 are not commutative, the final editing results at each site may be different. A divergence between the different sites exists. The figure below shows the tree structure of a very simple XML document at two editing sites (Site 1 and 2).

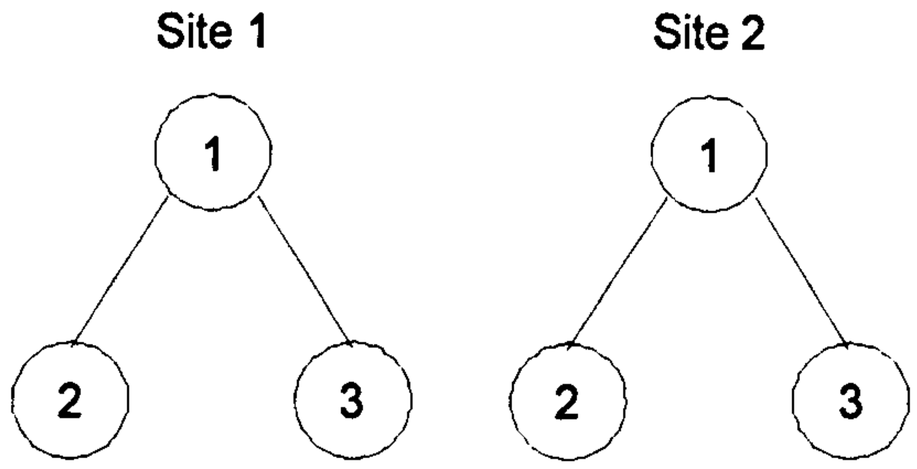

Figure 1.12: Divergence example: identical sites

Each of the two sites maintains a local copy of this document. At the beginning the tree structure at both sites is identical.

Operation O_1 creates a child node A below node 2 at the first position. Operation O_2 creates a different child node B also below node 2 at the first position. O_1 and O_2 are nearly concurrently executed on the local document copy before they are broadcast to the remote site. Due to the delay in execution the order of the operations O_1 and O_2 is different at each site. At site 1, the operation O_1 is executed before O_2. At site 2, the operation O_2 is executed before O_1. The results at sites 1 and 2 are divergent:

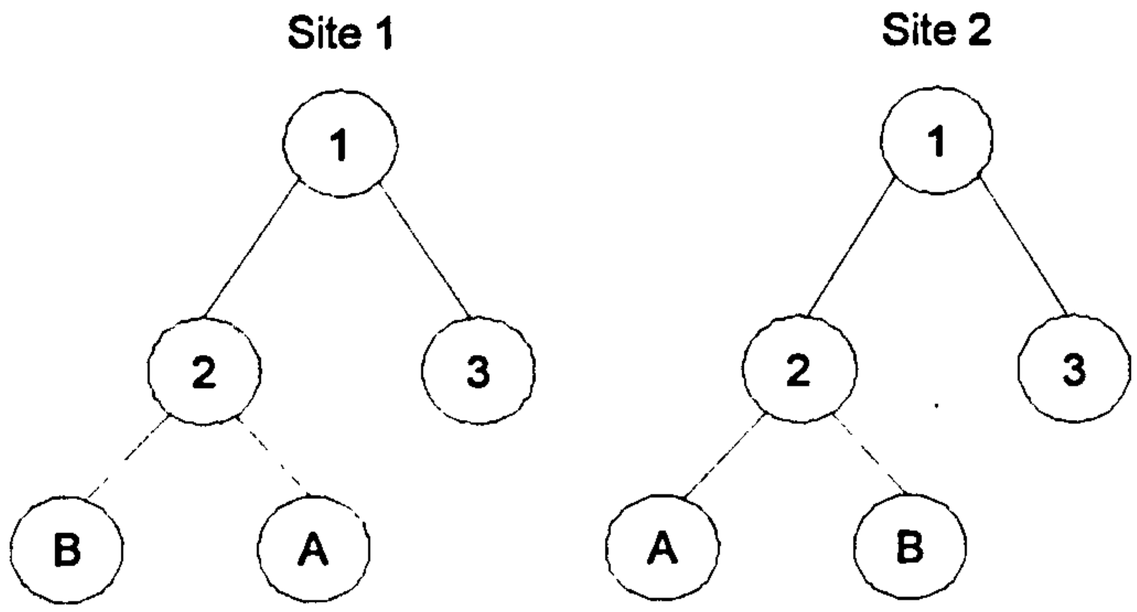

Figure 1.13: Divergence example: divergent sites

At site 1 operation O_1 is executed first and creates a child node A at the first position below node 2. Then O_2 is executed and creates a child node B at the first position below node 2. At site 2 the order of the nodes A and B is opposite to the order of site one after their creation.

1.5.2.2. Causality violation

As shown in the scenario operation O_3 is generated after the arrival of O_1 at site 2. If O_3 was created because of O_1 at site 2 then O_3 is dependent on O_1. This is also called a causal ordering relation between O_1 and O_3. Sun and Ellis (1998) define the causal ordering relation "$\rightarrow$" (following Lamport (1978)) as follows:

Definition 1.1: *Causal ordering relation "$\rightarrow$".*

Given two operations O_a and O_b, generated at sites i and j, then $O_a \rightarrow O_b$, if as follows:

1. $i = j$ and the generation of O_a happened before the generation of O_b, or
2. $i \neq j$ and the execution of O_a at site j happened before the generation of O_b, or
3. there exists an operation O_x, such that $O_a \rightarrow O_x$ and $O_x \rightarrow O_b$.

Definition 1.2: *Dependent and independent operations.*

Given any two operations O_a and O_b.

1. O_b is dependent of O_a if $O_a \rightarrow O_b$.
2. O_a and O_b are independent (or concurrent), expressed as $O_a \parallel O_b$, if neither $O_a \rightarrow O_b$, nor $O_b \rightarrow O_a$.

In the example scenario, O_3 is executed at site 3 before O_1. Thus O_3 at site 3 refers to a non-existent context, that is to be created by O_1. A user at site 3 would observe the effect of O_3 before observing the cause in O_1. This problem is called causality violation and should be prohibited for collaborative editing systems, where a real time synchronised interaction among multiple users is required. Causality violation in a tree structure can also lead to a state where one operation can not be executed because the operation it is depending on has not been executed yet. For example in the scenario shown O_1 is executed on site 2 and creates a node A below node 1 (see figure 1.14). The dependent operation O_3 is then executed on site 2 and creates a node B

below the new node *A*. At site 3 the operation O_3 is to be executed before O_1. This will result in an error state since node *A* at site 3 has not yet been created, thus node *B* cannot be created below *A*.

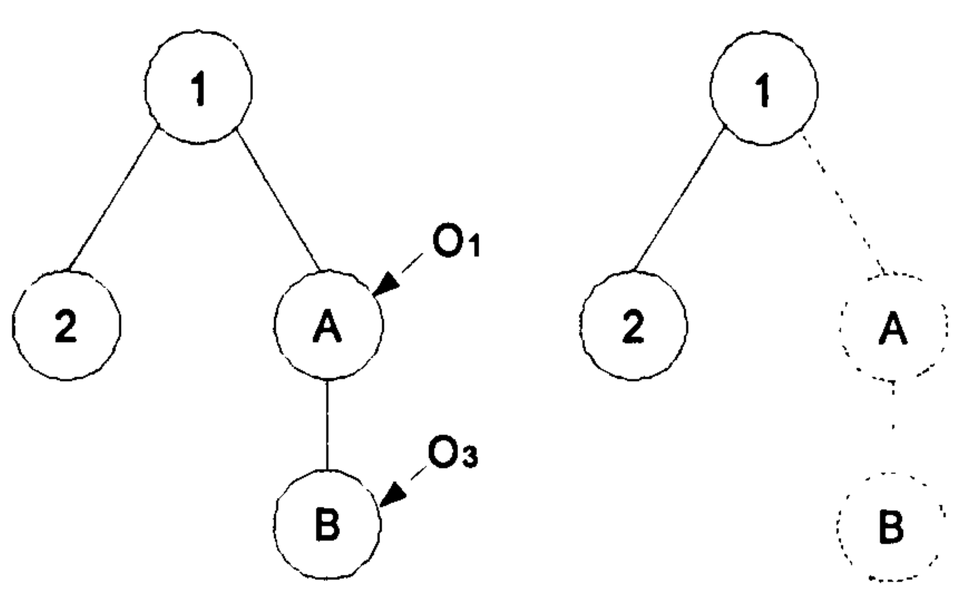

Figure 1.14: Causality violation

1.5.2.3. Intention violation

Intention violation occurs if the effect that was intended when executing an operation is somehow not achieved. Intention violation can occur when executing structural or mutational operations and a number of algorithms exist that try to preserve intentions in collaborative editing systems (see chapter 3). An example for an intention violation problem that is difficult to solve is as follows: If conflicting operations are generated to change the same attribute value or content of a node in an XML document, the effect of some operations may be lost. For example, the operations O_1 and O_2 are update operations on node *A* of a document. O_1 changes the value of attribute *a* of node *A* to *x*. O_2 changes the value of attribute *a* of the same node to *y*. Both operations are executed locally at the same time and then their change sets are sent to the remote sites to be executed there. The result is a different attribute value at each site and confused users. Since both operations are changing the same attribute to different values it is impossible (if the attribute only allows one value state) to accommodate their conflicting effects in the same target attribute. Either the attribute value is set to *x* or to *y* but not both. A consequence is that conflicting operations cannot be resolved automatically. Therefore, in a conflict situation, a consistent picture of what other users have done or of other users intentions is not guaranteed. Someone has to decide which attribute value is the correct one. But it may not be possible in all cases

to resolve their conflict. If convergence is somehow achieved then only one operations effect can be preserved and the other is lost.

The question arises: would the users execute their operations on the node, if they knew that another user is going to do a change on the same node as well? This is one point where awareness mechanisms can help to avoid conflicts.

1.5.2.4. Semantic inconsistency

In a collaborative editing environment, the inconsistency problems discussed earlier can lead to a syntactic inconsistency of the document. Existing algorithms used in real time collaborative editing systems solve the syntactic inconsistency problems, but they do not enforce semantic consistency. Ignat and Norrie (2002) give an example for the semantic inconsistency problem in a real time collaborative text editor: A shared document contains the text: "Helo everybody!". One user tries to correct the misspelled word "Hello" and inserts the letter "l", aiming to obtain the text: "Hello everybody!". Concurrently another user deletes the word "Helo" and inserts instead the word "Bye", in order to obtain the text "Bye everybody!". Depending on the algorithm that is used for consistency maintenance, the result is either a semantically inconsistent result or the change of one user is lost completely. Operational transformation (explained later) algorithms such as for example the GOT algorithm would result in following text: "Byel everybody!". As Ignat and Norrie state correctly, there is no automatic way to execute these conflicting operations and obtain a semantically consistent result. For the algorithm, the text "Byel everybody!" is correct. This is true with regard to the syntax. The GOT and other similar algorithms are fine-grained algorithms that work on character level. That is, the consistency is maintained in such a way that every character that is typed reflects in the document. In the natural language a character does not have a semantic value associated with it. This is the reason for the semantically inconsistent result "Byel". If the consistency maintenance algorithm would work on word level instead of character level, then a word like "Byel" could not possibly be created undeliberately. Ignat and Norrie give another example described in the paragraph below for semantic inconsistency, when the consistency maintenance algorithm works on word level. That is if all operations delete or insert whole words (Ignat and Norrie 2002):

The shared document contains the sentence: "The child go alone to school.". A user

deletes the word "go" and inserts "goes" instead, intending to obtain: "the child goes alone to school.". Simultaneously, another user inserts the word "can", changing the text into: "The child can go alone to school.". Unfortunately, after each user receives the operations performed by the other one, the result is: "The child can goes alone to school.". Here semantic consistency could not be enforced either. The conclusion that is drawn is that working at any level of granularity can result in semantic inconsistencies but working at a higher level usually translates into a more semantically consistent final result. Consistency maintenance algorithms today do not consider the context of a letter, word, sentence or paragraph. This is left to the users. The users have to know if a text is semantically correct and have to solve conflicts like the one above manually. Ignat and Norrie therefore propose an algorithm (treeOPT) that is based on a hierarchical structure of a text document to overcome the disadvantages presented above. This is done by allowing a change to the level of consistency maintenance as needed to either character, word, sentence or paragraph level.

It can be concluded that a hierarchical document structure has advantages over a linear document structure. This thesis therefore makes use of the hierarchical structure of XML documents and their properties, when it comes to consistency maintenance.

1.5.3. Concurrency control techniques

The next sections give an overview of the most common techniques for concurrency control. Concurrency control is a common method for prevention of inconsistency in database systems. Bernstein, Goodman et al. (1987) define concurrency control as follows:

"Concurrency control is the activity of coordinating the actions of processes that operate in parallel, access shared data, and therefore potentially interfere with each other."

Concurrency control techniques are used to maintain the consistency of a shared document. Two different types are discussed: techniques for conflict prevention and techniques for conflict resolution.

1.5.3.1. Locking

Locking is a concurrency control mechanism often used in database management systems to preserve serial equivalence and to synchronise access to shared data. The basic principle behind locking is quite simple but not very convenient for real systems. Thus optimised locking techniques and algorithms have been developed to address the real world problems of these systems. In the next section the two main locking strategies, pessimistic and optimistic locking, and some of the different variations are discussed.

Pessimistic locking

Emerged from synchronisation techniques used in operating systems (such as semaphores and monitors etc.), the oldest and easiest technique for concurrency control is the exclusive locking of data objects. In order to access a data object, an exclusive lock has to be acquired which excludes all other transactions from accessing the same data object. If a transaction T_2 wants to acquire a lock that is being held by another transaction T_1, T_2 has to wait until T_1 releases the lock. In the case of many parallel transactions accessing the same data objects, this can lead to a bad system performance. In order to improve the performance shared locks (or read locks) were introduced. The improved performance is based on the assumption that reading of data objects occurs much more often than writing. A shared lock therefore allows parallel transactions to read the same data object but not to modify it. In order to modify a data object, again, an exclusive lock has to be acquired.
This technique is called SX (or RX) locking. Figure 1.15 shows the compatibility of the two locking modes (Shared and eXclusive).

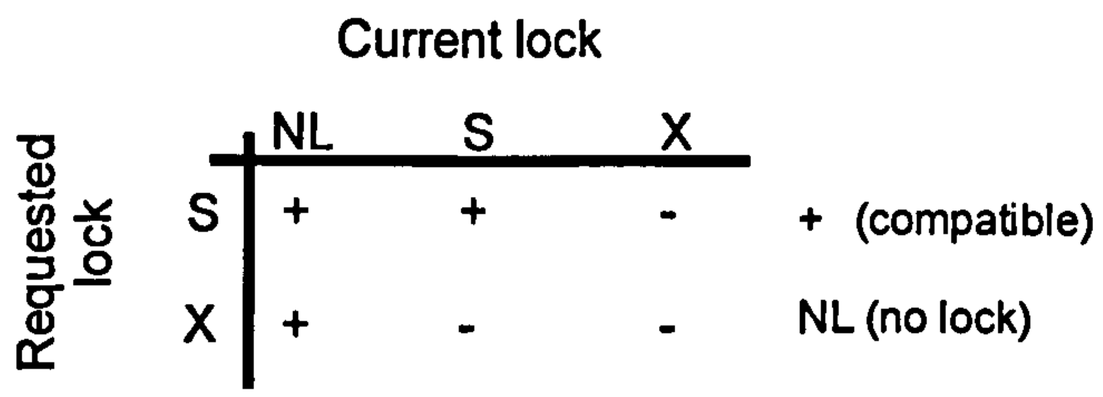

Figure 1.15: Compatibility matrix of SX-locking mechanism

NL (no lock) denotes the situation where a data object is currently not locked by any

transaction. A lock (shared or exclusive) is in that case always permitted. If another transaction holds currently a shared lock on a data object, another shared lock can be acquired on the same data object. Shared locks are compatible, that is a data object can be read by any number of transactions concurrently. A request for an exclusive lock will only be successful if the data object is currently not locked. If a lock conflict occurs, the requesting transaction is blocked and has to wait until the lock on the requested data object is released. This blocking can lead to a deadlock situation, where transactions wait for each other to release the lock on a data object and thus never finish. There are different methods for preventing and avoiding deadlocks. A prevention method is, for example, preclaiming or static locking, where all necessary locks are acquired before the beginning of the transaction. Methods for deadlock avoidance are, for example, immediate restart (no waiting) (Haerder 2001), wait depth limited (WDL) (Franaszek, Robinson et al. 1992) or the use of transaction time-stamps (wait/die, wound/wait) (Rosenkrantz, Stearns et al. 1978). These techniques are further discussed by Haerder (2001).

Another pessimistic locking technique is the so-called two phase locking (2PL). In the first phase (growing phase) a transaction acquires all necessary locks. In the second phase (shrinking phase) the transaction releases all the locks again (see figure 1.16).

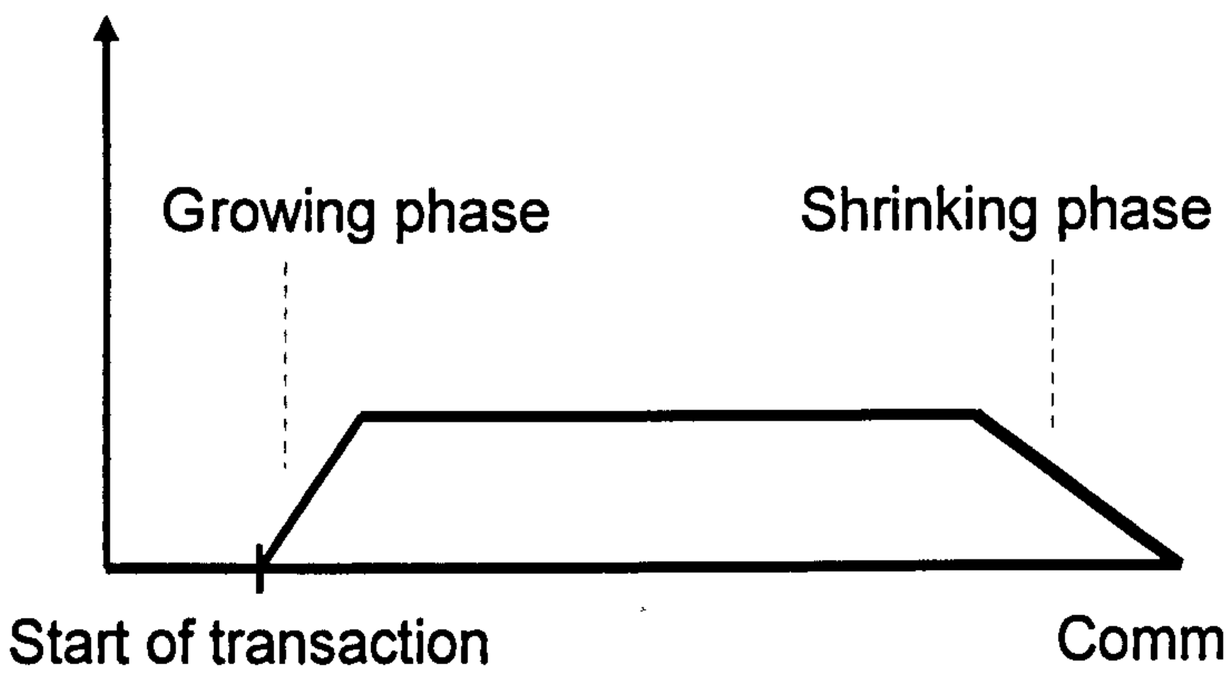

Figure 1.16: Two phase locking

If problems occur during the shrinking phase of a transaction and it has to be rolled back, a "dirty read" can occur (see section 1.5.1.2). One possible way of preventing a

dirty read situation is the so-called strict two phase locking. The complete shrinking phase is thereby performed at the end of the transaction when a successful completion of the transaction is guaranteed. Figure 1.17 shows a strict two phase locking transaction flow.

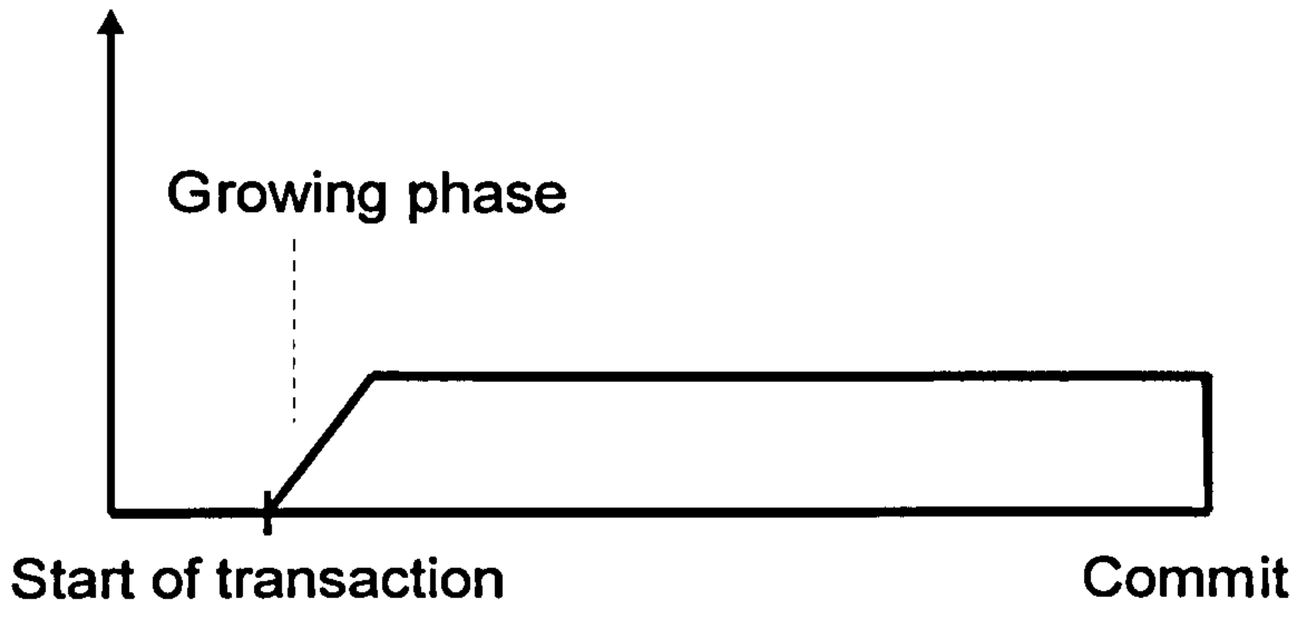

Figure 1.17: Strict two phase locking

Optimistic locking

The optimistic approach for concurrency control is based on the assumption that conflicting operations are rare and thus preventive locking of data objects would be an unnecessary high expense (Kung and Robinson 1981). The flow of transactions is not affected until these are committed, so nearly parallel work on the data is possible. At the end of a transaction it is determined if conflicting operations have occurred.
Following this technique a transaction is separated into three different phases :

1. In the read phase, data is read but write operations are performed on private copies of the data objects only.
2. After the transaction is committed the validation phase begins. In this phase it is determined that the transaction will not cause a loss of integrity or that it will return the correct result. If the validation fails, the transaction is backed up and started over again as a new transaction. In this case conflicts are resolved by rollback of one or more of the transactions involved. This means that more rollbacks occur than with pessimistic locking techniques. An advantage is, that deadlocks are not possible, in contrast to the pessimistic locking approach.

3. If it can be established during the validation phase that the changes which the transaction made will not cause a loss of integrity, the local copies are made global in the write phase.

The execution of operations on private copies of data objects has advantages and drawbacks. One advantage is that it yields a general approach to protecting other transactions from "dirty" read or write operations. A higher parallelism can be achieved due to the possibility of synchronous modification and reading of unmodified data objects.

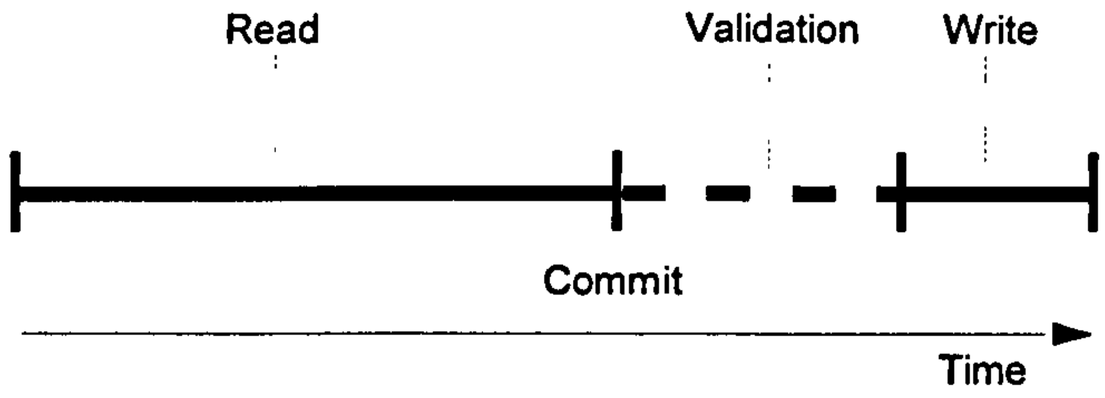

Figure 1.18: The three phases of a transaction

It is easy to realise the back up of transactions by merely discarding the changes on the private copies of the data objects. A disadvantage is that private copies of data objects lead to a higher memory demand of the system and require complex buffering mechanisms (Peinl 1987). The write phase can become very costly if the synchronisation granularity is small and thus a large number of write operations have to be executed. A problem is the danger of "starvation". This occurs if transactions constantly fail to validate. This can happen especially with very long transactions, because they have a large read-set and have to validate against many transactions. Additionally the late back up of a transaction causes a lot of unnecessary performed work.

It is possible to combine pessimistic and optimistic synchronisation strategies. Examples for such hybrid approaches can be found in database systems such as IMS Fast Path (Gawlick 1985) or GemStone (Butterworth, Otis et al. 1991).
The idea is to combine the advantages of both techniques. But the costs of realisation are very high and it is not guaranteed that the advantages of both approaches are ever achieved. IMS Fast Path uses a special combination of pessimistic and optimistic

synchronisation. Data objects are thereby processed unsynchronised and at the end of the transaction during the validation and write phase they are locked. Due to the reduction of the locking duration the probability of blocking other transactions is reduced considerably, compared with other pessimistic locking techniques.

A very similar approach is used in various version control systems such as the Concurrent Versions System[18] (CVS) and Subversion[19] (SVN). Changes on data objects are thereby made on the client side at first without locking. As soon as the changes are committed, the data objects are locked on the server. Now it is determined whether conflicts with other transactions exist (validation). By using timestamps it is determined if a data object has been changed since its transfer to the client. If validation fails, either the transaction is backed up and repeated or the client is notified of the conflict. This approach is called optimistic locking. The advantage of this technique is again the short locking duration during the validation phase. A problem is the relatively high probability of validation faults, so that this approach is mainly restricted to application areas with a relatively small number of concurrent users . This restriction is highly dependent on the locking granularity and with an accordingly fine granularity the probability of lock conflicts is extremely reduced.

Locking granularity

The locking granularity is a very important factor for the performance of systems that use a locking strategy for maintaining consistency. The locking granularity affects the number of lock conflicts as well as the administration effort. A coarse granularity, for example on single files, generally results in a higher probability of lock conflicts in a collaborative system. The advantage is that a transaction needs to acquire less locks and therefore the administration effort is relatively small. A finer granularity on the other hand produces a large administration overhead for maintaining a list of all current locks. The advantage is a low probability for lock conflicts and a high parallelism. Support for a flexible locking granularity (multi granularity locking (Bernstein, Goodman et al. 1987) or hierarchical locking) is a solution to these problems. In contemporary database systems the locking granularity varies from table locks, page locks to row level locking. Most of these systems support two

[18] Concurrent Versioning System (CVS): http://www.nongnu.org/cvs/, retrieved October 30, 2007

[19] Subversion (SVN): http://subversion.tigris.org/, retrieved October 30, 2007

or more granularity levels. Very few systems are able to lock single entries in a database table or even single characters. In most cases a very fine granularity is not necessary in database management systems in contrast to collaborative editing systems. The table below compares coarse and fine granularity locking strategies.

	Coarse granularity	Fine granularity
Probability of lock conflicts	High	Low
Number of locks to be acquired	Low	High
Administration effort	Low	High
Parallelism	Low	High

Table 1.6: Comparison of coarse and fine granularity locking

Working with a flexible granularity level can also help in maintaining the semantic consistency (see section 1.5.2.4). Ignat and Norrie (2003) propose the use of optional locking on different granularity levels in addition to their operational transformation approach for consistency maintenance. They propose five levels of granularity in a text document: document, paragraph, sentence, word and character. Each operation that is to be executed is then executed on one of these levels only. Depending on the type of document, the hierarchical depth can vary enormously. In this thesis therefore a more general disposition of granularity levels is proposed in the form of a node set. A node set is a partition of different disjunctive node types. A node type can be - depending on the used XML data model[20] (the logical representation of XML) - a document, an element, an attribute, a processing instruction, a text or a comment. In order not to use a specific data model a more general approach is: a node can either be of type root, child or leaf. In a document tree every node x, together with its child nodes, forms a node set with root node x. Every node y (that is no leaf node) owns the node sets of its child nodes. A leaf node does not own any child nodes.

1.5.3.2. Turn-taking

Turn-taking is a very simple method for consistency maintenance. With a turn-taking protocol, access to a shared resource is only granted to one, for example, user at a

[20] Different data models can be for example: XML infoset, Xpath or the Document Object Model (DOM).

time, so no inconsistencies can occur. The turn-taking protocol defines which user is the next to access the resource. This is either done by technical means (implemented in software) or by social protocols defined by the participants before their collaboration begins. This technique is similar to the pessimistic locking strategy, but much more restrictive, because it is always applied on the complete shared resource, for example, a document or a database. Pessimistic locking is usually applied only on parts of a shared resource, such as paragraph or a database table. Examples for collaborative applications that use a turntaking protocol are Microsoft's NetMeeting[21] and Hewlett-Packard's SharedX (Lauwers 1990). Concurrent editing is not supported by these systems, because there is only one active user at a given time.

1.5.3.3. Timestamp ordering

The basic timestamp ordering (Bernstein and Newcomer 1997) approach uses timestamps on data objects to preserve serial equivalence. The position of a transaction is thereby determined by the timestamp that is attached to it at the beginning of the transaction. Conflicting operations of different transactions have to be executed in the order of their timestamps. A transaction therefore has to see all changes of earlier transactions but must not see changes of later transactions. If these conditions are violated, the transaction is backed up and repeated again with a new timestamp. In order to validate these conditions, read and write timestamps are set for each read and write operation. The read timestamps (RTS) and the write timestamps (WTS) respectively thereby correspond to the last transaction that has changed the data object. A read operation of a transaction *T* with timestamp *ts(T)* on a data object *x* is not permitted if:

$ts(T) < WTS(x)$

This means that *T* is valid only if no later transaction than *T* has changed the data object. In the case of a write operation, transaction *T* is only valid if no later transaction has changed or read the data object. This means that a write operation of transaction *T* is not permitted if:

$ts(T) < Max(RTS(x), WTS(x))$

---

[21] Microsoft Corporation. 2000. http://www.microsoft.com/windows/netmeeting/, retrieved October 30, 2007

If one of these conditions is true, then the transaction accessing x is rolled back. Figure 1.19 shows a scenario, where transactions T_1 and T_2 access the data object x in the correct transaction order. However, when T_1 accesses data object y, this has already been changed by transaction T_3. The result is a rollback of T_1.

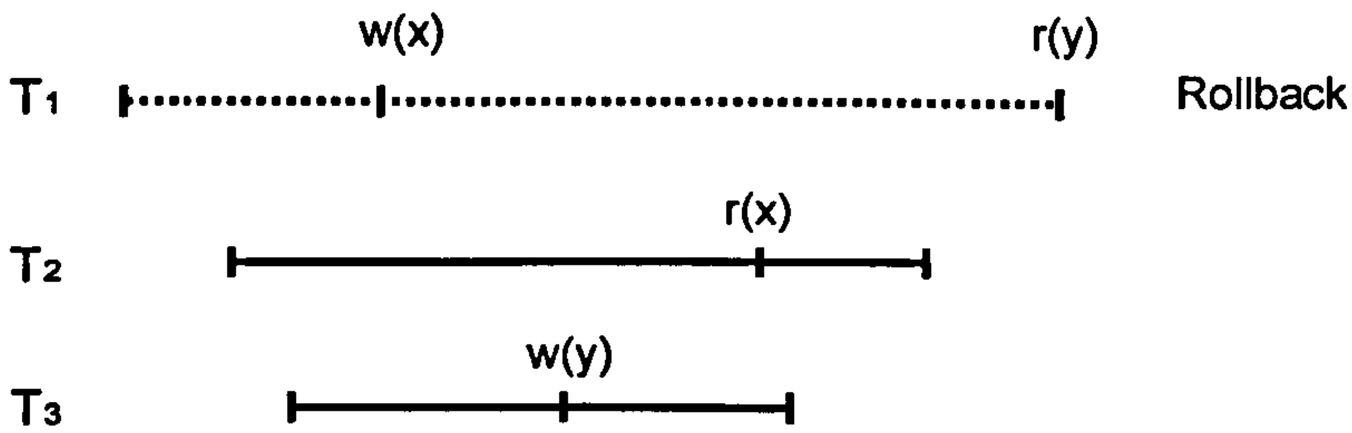

Figure 1.19: Timestamp ordering scenario

This shows that the probability of a rollback grows with the duration of a transaction because later transactions can access the same data objects. This means that in some cases a "starvation" of a transaction can occur. Another problem is the possibility of "dirty" writes due to uncommitted dependencies. However, timestamp ordering cannot solve the problems of intention violation and causality violation. Therefore, and because of the other problems, it is relatively unsuited for collaborative editing.

1.5.3.4. Causal ordering

With the causal ordering approach, operations may be generated and executed concurrently. Local operations are executed immediately after their generation. This leads to a good responsiveness. The execution order of the operations is constrained by their natural causal order and it is defined by using the vector logical clock (Fidge 1991). Some remote operations may be delayed until all causally preceding operations have been executed. This approach can achieve causality preservation only and does not address the problems of divergence and intention violation. Therefore, in many collaborative editing systems, this approach is used in combination with other concurrency control techniques. Operational transformation, for example, uses causal ordering additional to transformation of operations in order to achieve causality preservation.

1.5.3.5. Operational Transformation

Operational transformation (OT) is a technique for consistency maintenance by conflict resolution in collaborative editing systems. The main advantage of OT is that an operation that is generated is executed instantly without any delay. In contrast to conflict prevention techniques such as locking or turn-taking, the user does not have to wait until a lock can be acquired or a token is passed. In a collaborative editing system, operations are executed directly on a local copy of a document and are then distributed to the other sites and executed there. An operation that is received by a site may be transformed before it is executed on the local copy. The transformations are performed in such a manner that the intentions of the users are preserved and at the end the copies of the documents converge. Various operational transformation algorithms for collaborative editing applications that use a linear representation of the document[22] have been proposed:

dOPT, adOPTed (Ressel and Nitsche-Ruhland 1996), GOT , GOTO , SOCT2 (Suleiman, Cart et al. 1997).

Most of these algorithms follow the same principles for consistency maintenance. Therefore, in this section we will give a brief overview of these principles.

The operational transformation algorithm has been developed to oppose the problems of divergence, intention violation and causality violation. A collaborative editing system is thereby said to be consistent, if it always maintains the properties of convergence, intention preservation and causality preservation.

To achieve causality preservation, a time-stamping scheme based on a vector logical clock, is used. It allows the system to ensure that if an operation O_a "happened before" an operation O_b (see causal ordering relation "→"), then O_a is executed before O_b.

To achieve intention preservation and convergence, a total ordering relation "⇒ " (Raynal, Singhai et al. 1996) between operations is defined. The total ordering relation defines which operation has to be executed in which order at each site (see chapter 3 for a definition of the total ordering relation). Additionally all operations that are executed at each site are stored in a history buffer. Based on this total ordering relation and on the history buffer, a *undo/do/redo* scheme is defined. If a new

---

[22] The document that is to be edited.

operation O_{new} is received, all operations in the history buffer, that according to the total ordering relation, follow O_{new} are undone. This will restore the document state to before their execution. Then O_{new} is executed and after that all operations from the history buffer that were undone are now redone again. Additionally, the operations are transformed before their execution in order to cope with the modifications performed by other executed operations. To illustrate this, a collaborative editing scenario without transformational operation is shown in figure 1.20. There are two sites working on a shared document containing the text *"efect"*. Consider, that the text can be modified with the operation *Ins(p,c)* for inserting a character *c* at position *p* in the text. It is supposed that the position of the first character in the text is at position *p=1*. The users 1 and 2 generate two operations: $O_1 = Ins(2,f)$ and $O_2 = Ins(6,s)$ respectively.

When O_1 is received and executed on site 2, it produces the expected text *"effects"*. But when O_2 is received on site 1, it does not take into account that O_1 has been executed before it. The final result therefore is a divergence between site 1 and 2.

In order to obtain a correct result, operation O_2 needs to be transformed by taking into account operation O_1. If O_2 on site 1 is transformed into Ins(7,s) convergence finally is achieved.

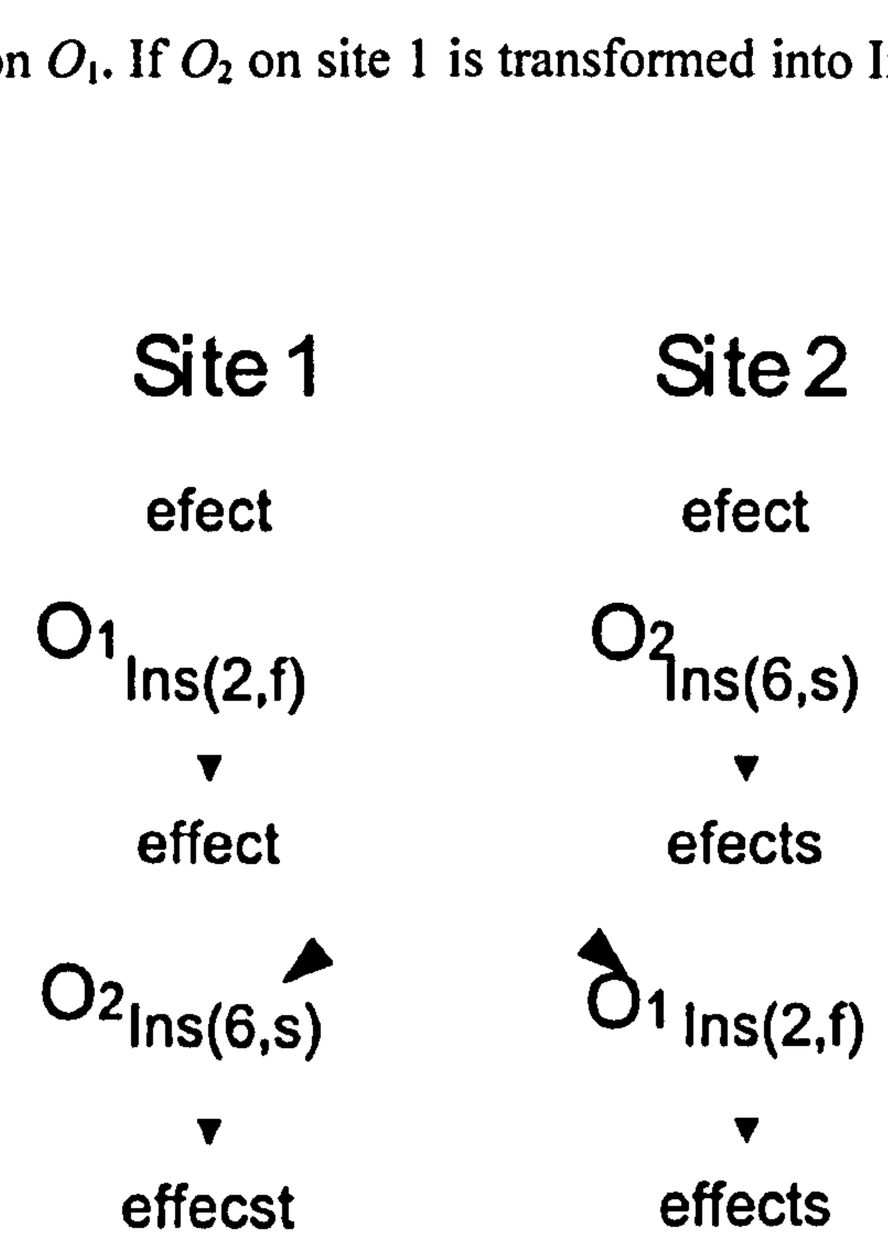

Figure 1.20: Incorrect integration of operations

Operational Transformation can only solve the problem of intention violation in resolvable conflictual operations such as the operations described above (Imine, Molli et al. 2003). It is not possible to solve non-resolvable conflictual operations using Operational Transformation. Non-resolvable conflictual operations are mutation operation targeting the same object and modifying the same attribute to different values whereas the attribute can only have one valid value. To solve this problem different approaches exist. One approach is to assign a priority to operations and execute only the operation with the highest priority. Another approach is not to solve the problem but to create a new version for the object and keep both operation effects. This means that at the end someone has to decide which version it the correct one. A third approach is to prevent the conflict by locking the object, in the case of a non-resolvable conflictual operation. In that case, one or more users have to wait until the lock is released, in order to perform their change of the objects attribute. This is the standard solution in database systems. Further methods of intention preservation are discussed in chapter 3.

1.5.4. Workspace awareness

"The primary role of awareness information is to make one's activity visible to others" (Dourish 1997).

Awareness mechanisms provide the opportunity for individuals to gain an understanding of the work of other group members and then use this information in order to co-ordinate activities across the group. Individual actions can be related to the activities of the group as a whole. The most important kind of awareness in collaborative editing environments is the workspace awareness:

"the up-to–the minute knowledge a person holds about another's interaction with the workspace" (Gutwin, Roseman et al. 1996).

This includes knowledge about who is in the workspace, where they are working, what they are doing and what they intend to do next. Workspace awareness reduces the effort needed to co-ordinate tasks and resources, helps people to move between

individual and shared activities, provides a context in which to interpret utterances and allows anticipation of others' action. While most operational transformation and locking algorithms only support maintaining the syntactic consistency of a document, awareness mechanisms can also help maintaining semantic consistency. For example imagine two users working on a book. User one edits chapter five while at the same time the second user works on chapter one. Now the first user wants to refer to a section of chapter one, that is currently being edited by user two. If user one does not know what user two is exactly working on, he might refer to a section that is not finished at that moment in time. Thus if changes are made to that section, the reference might be semantically incorrect. If user one knows exactly what user two is working at, he probably would wait until user two is finished with that section before he references it. Awareness can help in this case by providing the means for user two to make others aware that he is still working on a certain section and that it is not finished. Usually user one would call user two and ask if he is finished with that section or not. Or he would write an email or use an internet chat to communicate with the other user. These kinds of awareness mechanisms disturb the workflow when editing a document. It would be much more efficient, if each user could see at one glance who is working on which part of a document. Also if he could make others aware of what part he is working on and without much trouble. Thus it is important to integrate awareness mechanisms into a collaborative editing system so that they effectively support the editing process without impairing it. One approach is to augment the collaborative editing interface with new components – widgets – that show some of the missing information about other collaborators (Baecker, Nastos et al. 1993). Examples for awareness mechanisms are as follows:

- Teleporting
- Telepointing
- Multi-user scrollbars
- What you see is what I do (WYSIWID) views
- Miniature views
- Radar views

"Teleporting" is a fast way of looking at an other person's part of the workspace by, for example, showing a screenshot of the other person's view. "Telepointing" allows to see another person's mouse movements. A multi-user scrollbar shows each person's relative location in the workspace. The WYSIWID view provides full-size detail of another person's interaction. The miniature view shows an overview of the entire workspace and the radar view presents additional information about others' locations. Incorporating all these different widgets into one collaborative editing system can make it look complicated and rather confusing for a user. Thus the awareness mechanisms chosen need to be integrated in a user friendly way and should only provide important information.

Integration of good awareness mechanisms can contribute a lot to the usability of a collaborative editor. This problem is very rarely addressed by contemporary collaborative editors.

1.6. System architectures

This section briefly discusses the different software architectures that are used in contemporary distributed systems. Each software architecture has its assets and drawbacks. When developing a collaborative system it is important to take the different architecture properties into consideration.

1.6.1. Centralised architecture

The centralised architecture is the most common type of a client/server architecture. A centralised server (e.g. database management system) maintains the data that is to be shared between multiple clients. The clients cannot access the shared resource directly. Access to the data objects is only possible via the server. A server can have one or multiple server processes. If a server has only one server process, then concurrent access to a shared resource is not possible. Hence no inconsistency problems occur. The disadvantage is the low system performance, because only one user can access the data objects (for example a shared document) at a time. Systems that use this kind of architecture provide the illusion of concurrent access by supporting only very fine grained operations which can be executed in a very short time. A common approach in database systems is a server with multiple server processes or threads

which can access the shared resource concurrently. This increases system performance dramatically but raises the problem of possible inconsistencies which needs to be tackled by the mentioned concurrency control methods. Another disadvantage of this architecture is that the local response time may be long.

This is due to the fact that every operation is

- first sent to the server, then
- the server executes it and sends a message to all clients to inform them of the update and then
- the client receives the reply message from the server and updates the local user interface.

This process is quite time consuming and the speed of completion for these steps is highly dependent on the network latency. A high network latency (for example in the internet) results in a bad system performance. Figure 1.21 shows a centralised architecture.

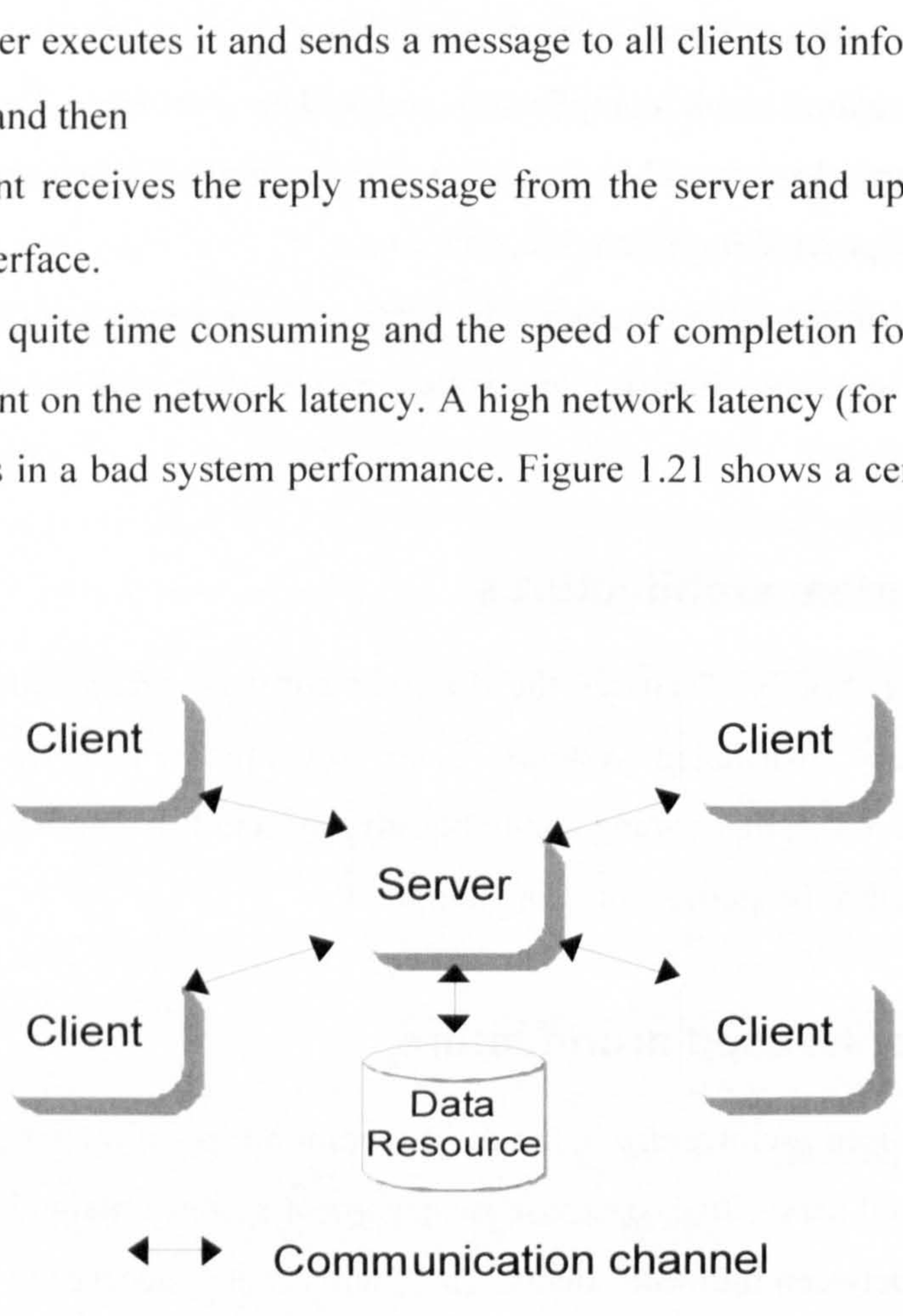

Figure 1.21: Centralised architecture with single server process

1.6.2. Replicated architecture

With a replicated architecture there is no central data resource and server. Each client site holds a replicate of the server process and the shared data resource. The server process at each client site is responsible for maintaining the consistency of the replic-

ated data and thus needs to communicate with all the other sites. This architecture is shown in Figure 1.22. A replicated architecture can provide fast response times with optimistic execution. When a local operation is generated it is executed immediately by the local server process. The result is instantly visible at the local client because no network communication or validation process is necessary. The response time therefore, is independent of the network latency. After the operation is executed locally it is propagated to all other sites that are interested in the update. When the other server processes receive the change operation, they execute the received operation on their local copies of data objects. A disadvantage of this approach is that inconsistencies on replicated copies may occur due to concurrent update to the shared document made by multiple server processes.

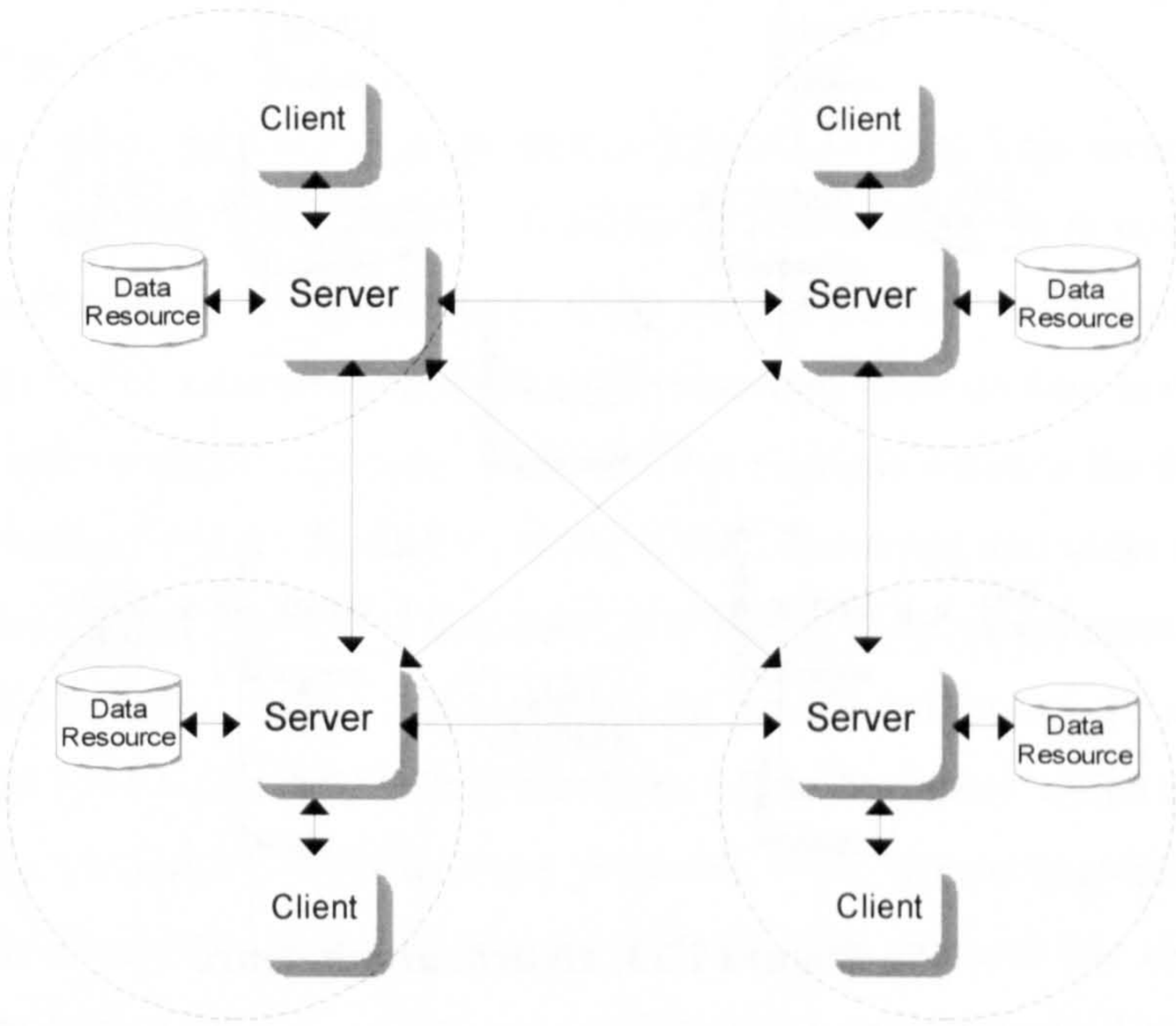

Figure 1.22: Replicated architecture

1.6.3. Hybrid architecture

The hybrid architecture is a mixture of both the centralised and the replicated architecture. With the hybrid architecture, each site holds a client and a server process. Each site also holds a partial or complete copy of the shared data resource. In addi-

tion there is also a server site holding the shared data resource and a server process. In general this architecture works in the following way:

There are two different types of operations with the hybrid approach; problematic and non-problematic. Operations that do not cause inconsistency when executed concurrently (for example create operations) are non-problematic and operations that may cause inconsistency (for example delete and update) are problematic operations. When non-problematic operations are generated they are executed locally before being sent to remote sites for execution. This produces good response times. The other type of operations need to be executed via the server site to avoid inconsistency. The response time for these types of operations is still dependent on the network latency.

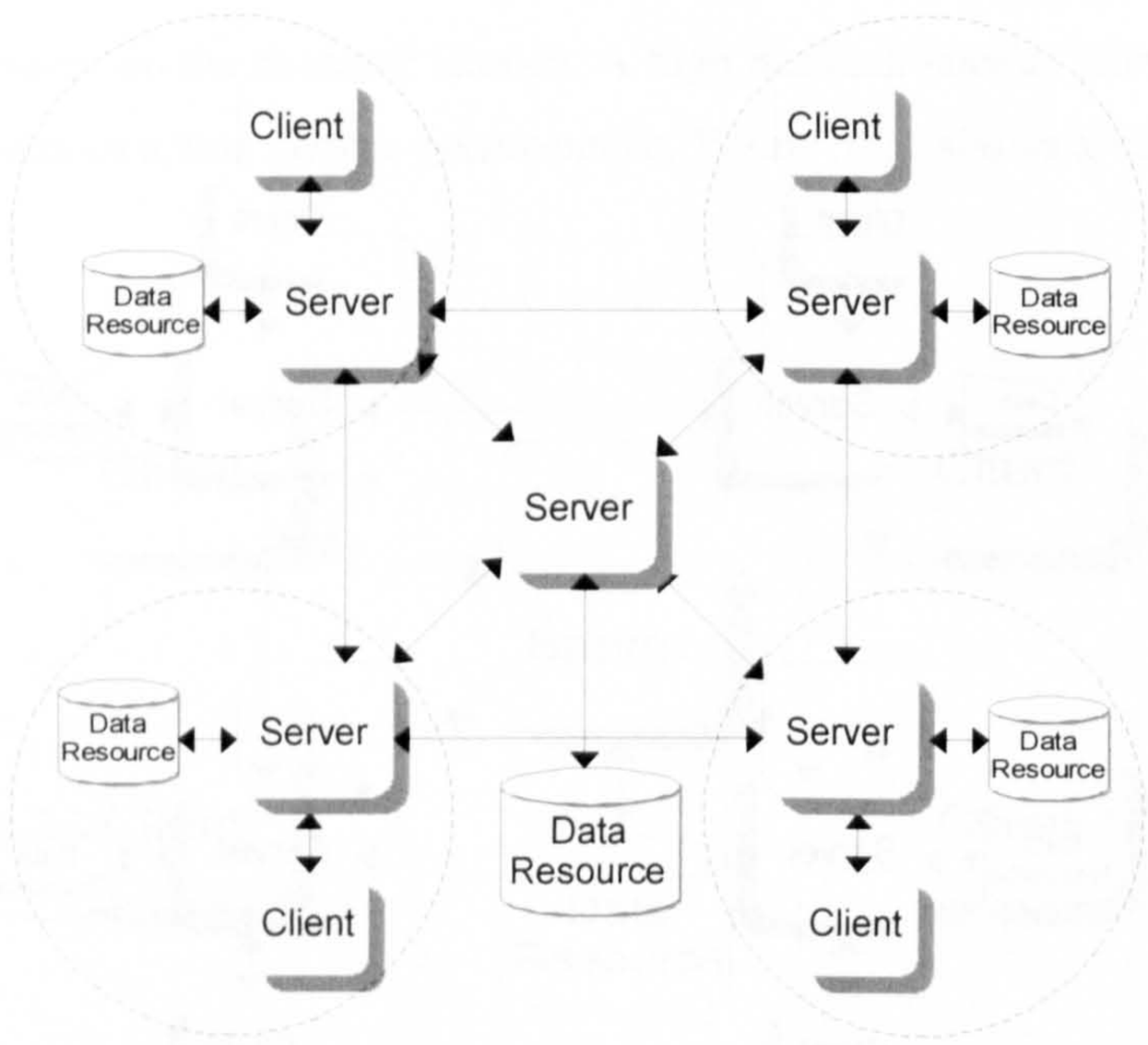

Figure 1.23: Hybrid architecture

Besides the better responsiveness, the advantage is that there is always a central site that holds the current correct version of the document. A copy of this version can be obtained, for example, if a new site joins the session.

1.7. Web-based editing

The Internet made it possible for an author to quickly and easily publish a document and make it accessible to a large group of readers independent of where they might be located. Today there is a trend, where this model of one publisher and many readers is shifting towards a more interactive model, where multiple authors interact. An example for this is the vision of the Semantic Web, where agents create, update and peruse information freely. A more concrete example is the WebServices architecture from IBM, where the notion of "author" and "reader" is replaced by peers that interact across the Web. The HTTP extension WebDAV (Web Distributed Authoring and Versioning) allows collaboration of several authors over a Web resource and Wikis are normal HTML pages that have hyperlinks enabling any reader to edit them at any time. They serve as places in the web, where knowledge is stored and generated by a collaboration of users.

All of these tools support coarse-grained collaboration only. This means that different authors can only edit different documents concurrently. It is not possible for multiple authors to edit different parts of the same document at the same time. They do not support fine grained collaboration, because they have no way to maintain consistency within a shared resource. WebDAV for example extends the HTT-Protocol in a way, that one author can lock a certain HTML document and make changes to it, but no other user is allowed to change it at the same time. Awareness mechanisms (where each author is aware of the rest of the group's activities) are not supported either. They have no notion of other instances of the document which should reflect remote edits. One of the most important protocols in the internet (besides TCP/IP and UDP) is the Hypertext Transfer Protocol (HTTP). This protocol has certain properties that are important if it is used in a collaborative environment. The next section discusses this protocol and it's properties.

1.7.1. HTTP

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP has been in use in the World-Wide Web since 1990 and the Internet Engineering Task Force (IETF) proposed the Hypertext Transfer Protocol as an Internet standard in 1992. Without this

standard, the World Wide Web as we know it would not be possible. The standard allows every computer that is connected to the internet to communicate with any other computer that provides information based on this standard protocol. To access this information a so-called Uniform Resource Identifier (URI) is used. This URI unambiguously identifies any resource in the Internet. The first version of HTTP, referred to as HTTP/0.9, was a simple protocol for raw data transfer across the Internet. HTTP/1.0 as defined by RFC 1945 (Berners-Lee, Fielding et al. 1996), improved the protocol by allowing messages to be in the format of MIME (Multipurpose Internet Mail Extensions) like messages, containing meta-information about the data transferred. The latest protocol version HTTP/1.1 additionally provides methods to indicate the purpose of a request. This is important for applications that require more functionality than simple retrieval of information, including search, front-end update and annotation. What is important to mention is, that HTTP is a stateless protocol. This has some effect on the applications that use this protocol to transfer data. The HTTP protocol is a request/response protocol. A client sends a request to the server and the server responds with a success or error code and the information body that has been requested. After this request and response, the connection between the client and server is disconnected and the server and client "forget" about the past request. The protocol knows no notion of session or state. This information is important for collaborative editing systems that work with a replicated or hybrid architecture, where each client and server needs to have information about other participants in the collaboration. Another problem is that HTTP requires to open a new TCP connection for every request. This causes an unnecessary overhead and network traffic. To solve these problems, new protocols were developed, such as the Session Control Protocol (SCP). This protocol allows a server and a client to have multiple conversations over a single TCP connection. Another approach to allow different collaborating sites to be informed about the collaborations participants is the experimental multiplexing protocol SMUX. SMUX is a session management protocol separating the underlying transport (HTTP) from the upper level application protocols. Neither of these techniques was accepted. Other application dependent protocols store information about the request in order to map a following request to a running session. These protocols are not standardised and depend strongly on the

used technology, when developing a web application. But these proprietary protocols are mainly used for session control today. This problem of session control therefore needs to be addressed when developing a collaborative system that uses HTTP as a transport protocol. The main advantages of HTTP are that it is a widely accepted standard, has a good performance, was especially developed for hypertext documents and can be used to transfer data even through firewalls.

1.8. Summary of contributions and thesis outline

This thesis focuses on developing a system for collaborative editing of XML documents. The following new contributions to the CSCW field are made by this work.
A framework is developed that enables synchronous collaborative editing of XML documents, supports different awareness mechanisms and allows XML applications to be extended with the collaborative editing feature. Specialised editors for any kind of XML document type (e.g. SVG, X3D, DocBook) can use the framework to enable them to work collaboratively on documents. The framework is called Collaborative Editing Framework for XML (CEFX). Further contributions to CSCW are as follows:

- A flexible plug-in architecture allowing third party developers to extend CEFX with new awareness widgets, concurrency control mechanisms and conflict resolution schemes. This allows the framework to perfectly adapt to the requirements of the process workflow. Applications that use CEFX will profit from new third party developments as well, without the necessity of changing source code.
- Support for collaborative work using heterogeneous applications using the same XML document type. This is especially helpful, for example, in the automotive industry where different tools are used to manipulate, analyse and check the same data.
- A flexible consistency maintenance algorithm for XML documents supporting optional locking of document parts. This supports adapted working schemes and facilitates privacy support.
- A novel approach to the extension of single-user applications by making use of

the Document Object Model (DOM) as a common interface and Aspect-Oriented Programming (AOP). The CEFX approach simplifies the integration of the collaborative framework into an existing single-user application supporting both, the collaborative-aware and the collaborative-transparent approach.

Chapter 2. Structure and conflict probability of XML documents

To develop a system for collaborative editing of XML documents, a concurrency control (also called synchronisation) algorithm that fits to the XML data model is required. Contemporary concurrency control algorithms for real-time collaborative editing systems are designed for their special application area and data model. Most of the existing concurrency control algorithms support linear data models only. In order to find an optimal concurrency control algorithm for editing XML documents, it is necessary to understand how conflicts can occur in a collaborative environment. If multiple persons work collaboratively on a simple text document, each structural change (e.g. deletion or insertion of a character) causes a conflict.

In case of XML documents this is different. A structural change merely leads to a conflict, if two or more persons change the same part of the document concurrently. The reason for the differences in the conflict probability lies in the structure of the data. A better understanding of the coherence between the structure of a document and the conflict probability helps to find an optimal concurrency control algorithm. Thus in the next sections the conflict probability in case of multiple persons collaboratively working on an XML document is being investigated.

2.1. Analysis of XML documents

The probability of conflicts depends on many different factors. Among these is the structure of the document itself. Analysing the structure of XML documents can lead to a better understanding of when and why conflicts occur. In this section the structure of XML documents is analysed.

As one representative of XML documents, Scalable Vector Graphic (SVG) documents are analysed. SVG is an XML markup language for describing two-dimensional vector graphics, both static and animated. It is an open standard created by the World Wide Web Consortium (W3C). Specialised graphics tools are used for drawing the graphics which are then stored in the SVG document format. Because graphic documents are often manually created (i.e. not generated by a computer)

SVG is a candidate document format for collaborative editing. SVG can be used for example in multimedia, documentation (e.g. figures), presentations (e.g. charts), architecture (e.g. constructional drawing) and engineering (e.g. circuit diagrams); basically anywhere where graphics are used.

Analysis of XML documents means in this case the identification of typical XML document properties and analysing a number of XML documents towards the occurrence and values of these. The following XML document properties identified are relevant for the analysis:

- Number of elements
- Number of hierarchy levels
- Number of elements per hierarchy level
- Number of child elements below an element (branches)
- Number of references between elements

Other XML document properties that are not relevant for the probability of conflicts are properties that have no influence on the overall structure of a document. These are for example:

- Total number of attributes
- Number of attributes per element
- Document Type Definition or XML Schema
- Comments
- Namespace definitions

The Document Type Definition or XML Schema can only indirectly influence the conflict probability by defining a necessary coherency between nodes and thereby influencing the structure. XML documents basically consist of a number of elements that are linked to each other as a tree graph. The figure below shows the structure of an example SVG XML document.

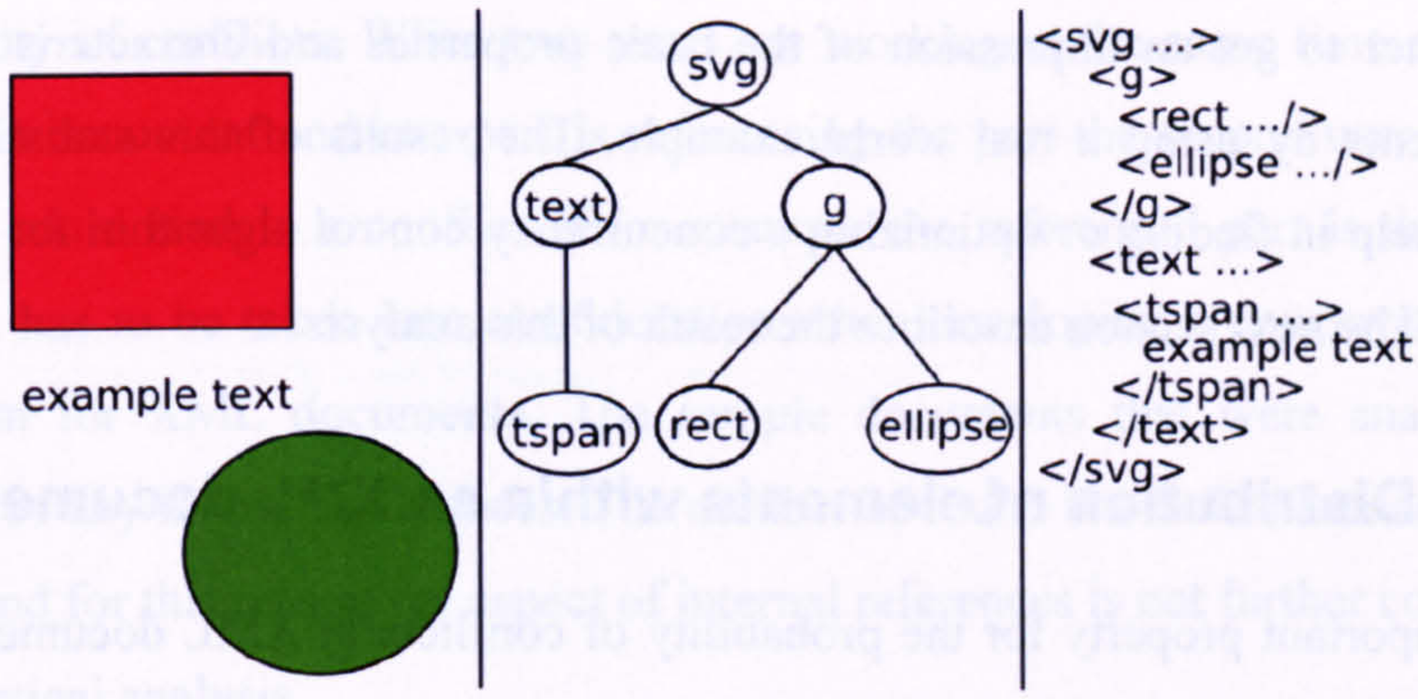

Figure 2.1: Example SVG graphic, XML document structure and source code

In order to analyse the structure of SVG documents, a computational technique was developed that generated a statistic on the properties of the XML documents.
The computational technique operates on a set of XML documents by analysing the documents with respect to the properties stated above and generating a character separated values (CSV) file containing the collected information. This data can then be used to generate charts that visualise the analysis results.

A total number of 362 different SVG files were analysed. 209 of these SVG documents contain circuit diagrams of a car manufacturer located in Stuttgart. These documents were chosen, because of their importance in the manufacturing process and because they were manually collaboratively created by a large team of technicians and engineers. The method by which the circuit diagrams are produced is by a collaborative turn-taking process. This is why they are especially interesting for this analysis. Enabling collaborative editing of the SVG circuit diagrams in real-time would increase significantly the efficiency of the whole production process of the car manufacturer. The other 153 SVG files are collected from the Internet and other sources. They have no common usage or purpose. Those were chosen in order to have a broader variety of other kinds of SVG files as well, not only circuit diagrams.

The number of SVG files scanned is too small to be statistically representative for SVG documents in general. The intention of this analysis is not to generate a statistic that is representative for all possible SVG or XML documents - this probably would not be possible because of the vast number of possible formats and characteristics -

but rather to get an impression of the basic properties and characteristics of XML documents by using a real world example. The results of this statistical analysis could help in finding or optimizing a concurrency control algorithm for XML documents. The next section describes the result of this analysis.

2.2. Distribution of elements within an XML document

One important property for the probability of conflicts in XML document editing is the overall number of elements in a document.

But of even more importance than this is where these elements are located and how they are arranged within the document.

Therefore the analysis software, used in this analysis, generated a file containing the following information about the analysed XML documents:

- Total number of elements
- Total number of attributes
- Maximum number of attributes per element
- Average number of attributes per element
- Number of hierarchy levels
- Number of references
- File name
- Document type

The total, the maximum and the average number of attributes is not relevant for the probability of conflicts, because it does not influence the tree structure of the document. Anyhow this information could be of interest in order to understand the importance of attributes for the SVG document format. This, and because it was easy to implement in the analysis software, are the reasons why this information is collected additionally. The collected information shows that attributes are very often used in SVG documents. Each element in a SVG document contains between zero and eight attributes. The average number of attributes per element is three.

The number of references between elements in a document could be relevant for the

probability of conflicts. When two users are working at the same time on different parts of a document and one part is referencing the part the second user is working on, this could lead to a conflict if for example the referenced part is deleted. This problem has to be taken into consideration when developing a concurrency control algorithm for XML documents. The sample documents that were analysed here, hardly had any internal references. The number of found internal references was negligible and for this reason the aspect of internal references is not further considered in the statistical analysis.

Here is a sample from a generated information file with the properties listed above for each document in one row. Each property is thereby separated by a pipe ("|") character:

```
5516|11001|8|2|7|0|funktionsflow_sgsystemplan80.svg|svg10.dtd
26|71|7|2,72|4|0|funktionsflow_sgsystemplan35.svg|no DTD
3855|13334|8|4,5|5|0|schaltplan17.svg|svg10.dtd
255|705|8|1,727|5|0|funktionsflow_sgsystemplan50.svg|no DTD
789|1680|8|3|6|0|funktionsflow_sgsystemplan84.svg|svg10.dtd
1010|2971|7|3|5|0|funktionsflow_sgsystemplan119.svg|svg10.dtd
305|630|8|1,757|5|0|funktionsflow_sgsystemplan40.svg|no DTD
387|925|8|2,878|4|0|funktionsflow_sgsystemplan145.svg|svg10.dtd
584|1256|8|2,961|6|0|funktionsflow_sgsystemplan114.svg|no DTD
```

As can be seen in the above sample, the property values of each document can be very different. Some documents are large (contain many elements), some are quite small. The number of attributes can vary a lot and the number of hierarchy levels span from one to nine in the analysed documents. The filename and document type information is not important for the statistical analysis but is also contained in the generated information file in order to be able to know exactly which documents were analysed and what document type they have.

The first step of this analysis was to determine the typical size of a SVG document. The size of a document is very important for the probability of conflicts. If a document is relatively small the probability for a conflict is higher than with large documents under certain conditions. Within the number of sample documents that were analysed it was found that the size of the documents vary a lot. This means it is quite difficult to make a general statement on the size of SVG or XML documents.

Figure 2.2 shows the document sizes of the sample documents. The documents were grouped in size categories of 250 elements.

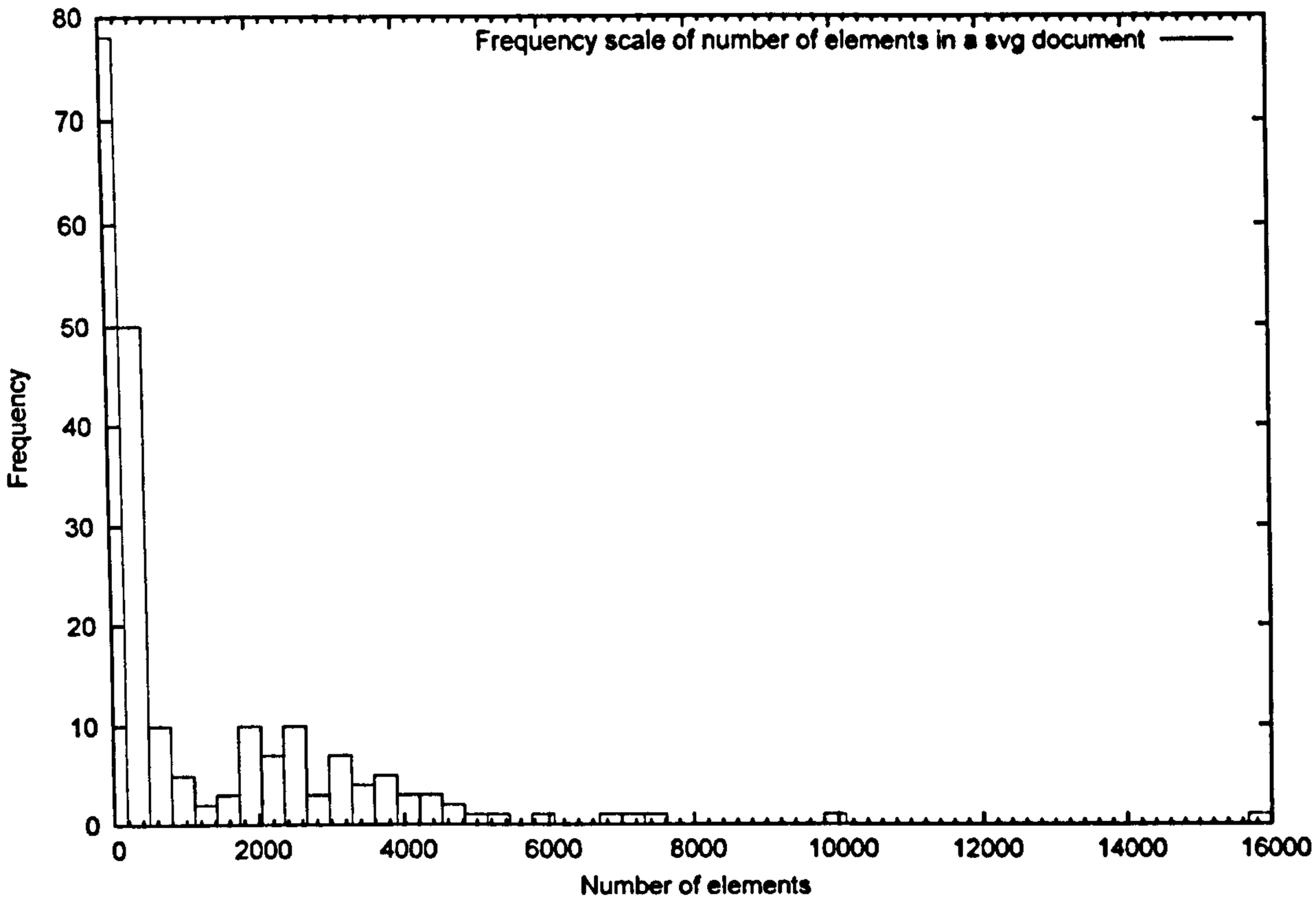

Figure 2.2: Frequency of documents within a certain range of number of elements

As stated above not only the size of a document but also the document tree's topology is important. This information was also collected and a second set of derived properties was generated that contains the information on the number of elements per hierarchy level for each document. Below is a sample from the generated data. On hierarchy level 0 there is always only one element, the root element (or root node). Each hierarchy level in the file is separated by a pipe ("|") character:

```
1|6|273|2061|2144|957|74
1|2|12|11
1|8|1528|2263|55
1|3|175|72|4
1|7|8|491|193|89
1|4|126|819|60
1|3|125|107|69
1|2|376|8
1|2|351|168|35|27
```

Figure 2.3 shows a sample XML document tree and some of its structural properties.

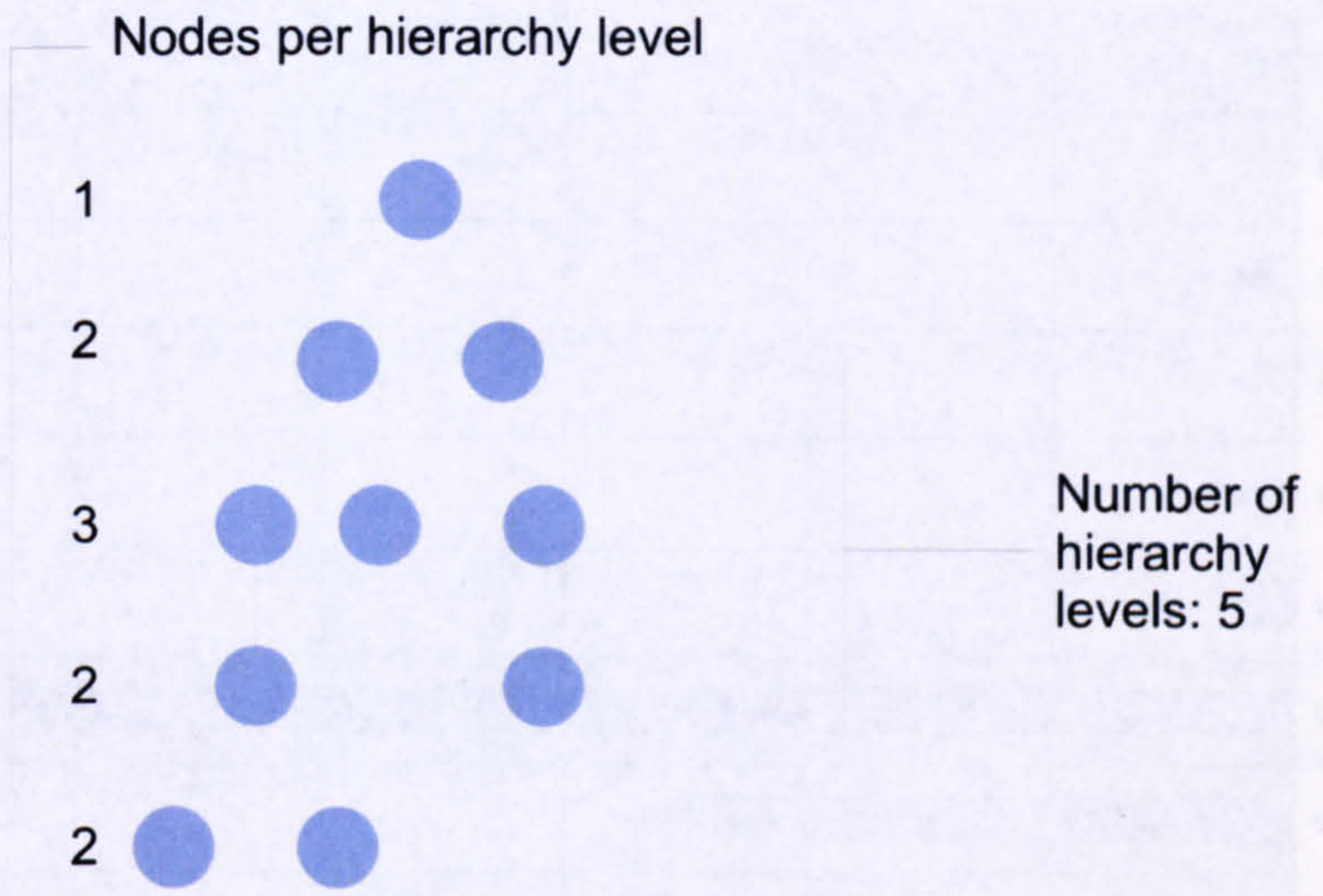

Figure 2.3: Example for analysed properties

Next to the number of nodes per hierarchy level, the number of child nodes per node (i.e. the links between the nodes) has an influence on the conflict probability. This issue is further discussed in section 2.3.1. The tree topology as well as the overall size of an XML document can vary very much depending on the document type. As mentioned, in this case graphic documents (SVG) were analysed. Other document formats such as text or multimedia documents can, for example, in average have many more or less elements than the ones analysed here. This means, as mentioned

before, that the results are not to be seen as generally valid for XML documents.

Figure 2.4 shows a diagram of the average distribution of elements on the different hierarchy levels of the scanned files. It shows the average number of elements on the X-axis and the hierarchy levels 1 to 9 on the Y-axis. Most of the elements of the scanned SVG documents can be found on hierarchy level 2 and 3. Hierarchy level 0 always contains one element, the root node.

In order to get a more general view of the structure of the sample documents, the median (instead of the average) of the number of elements is calculated per hierarchy level. This is done to eliminate peaks of values in a small number of documents.

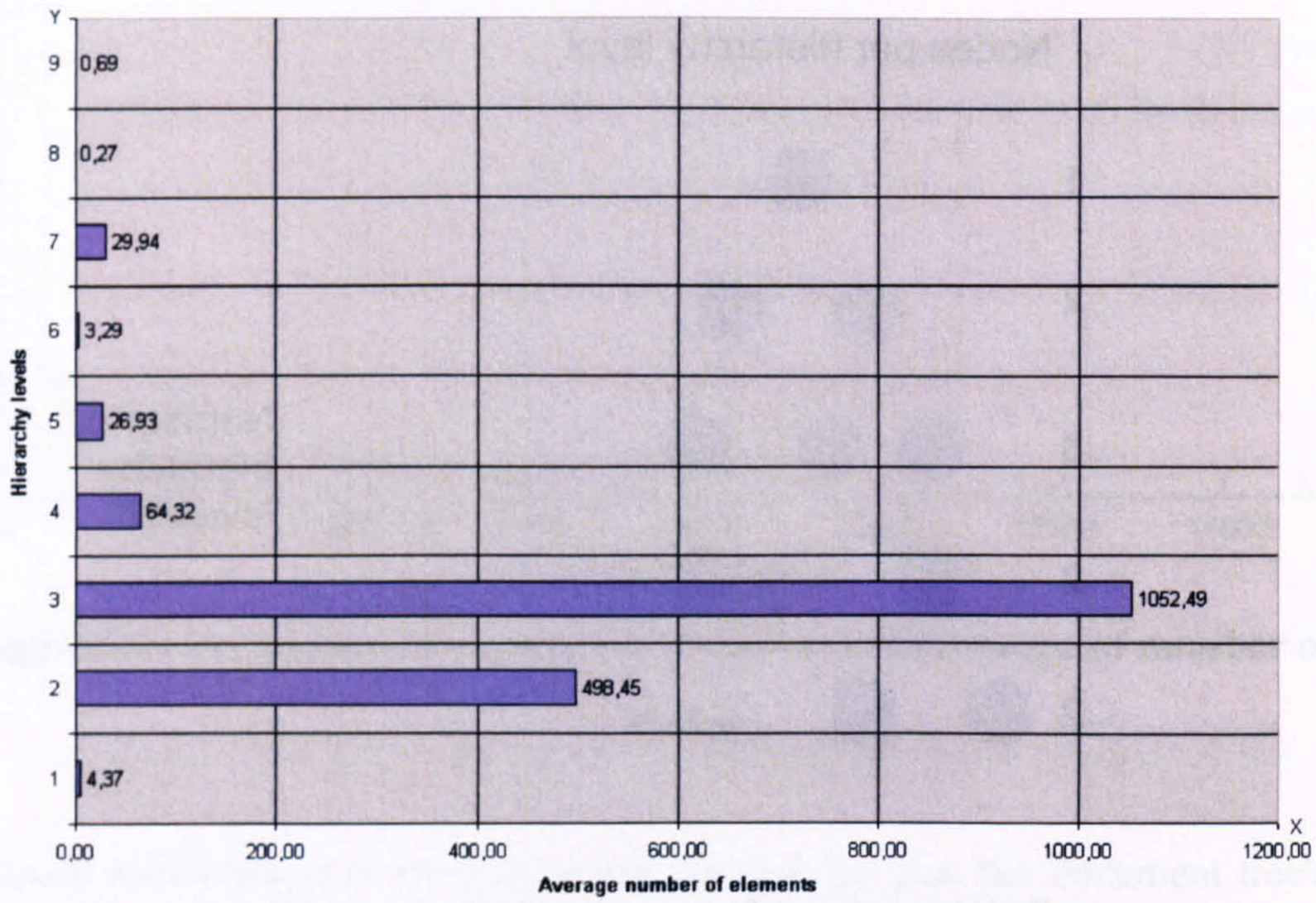

Figure 2.4: Average distribution of element on hierarchy levels

Figure 2.5 shows the distribution of elements on the hierarchies of the sample documents by median.

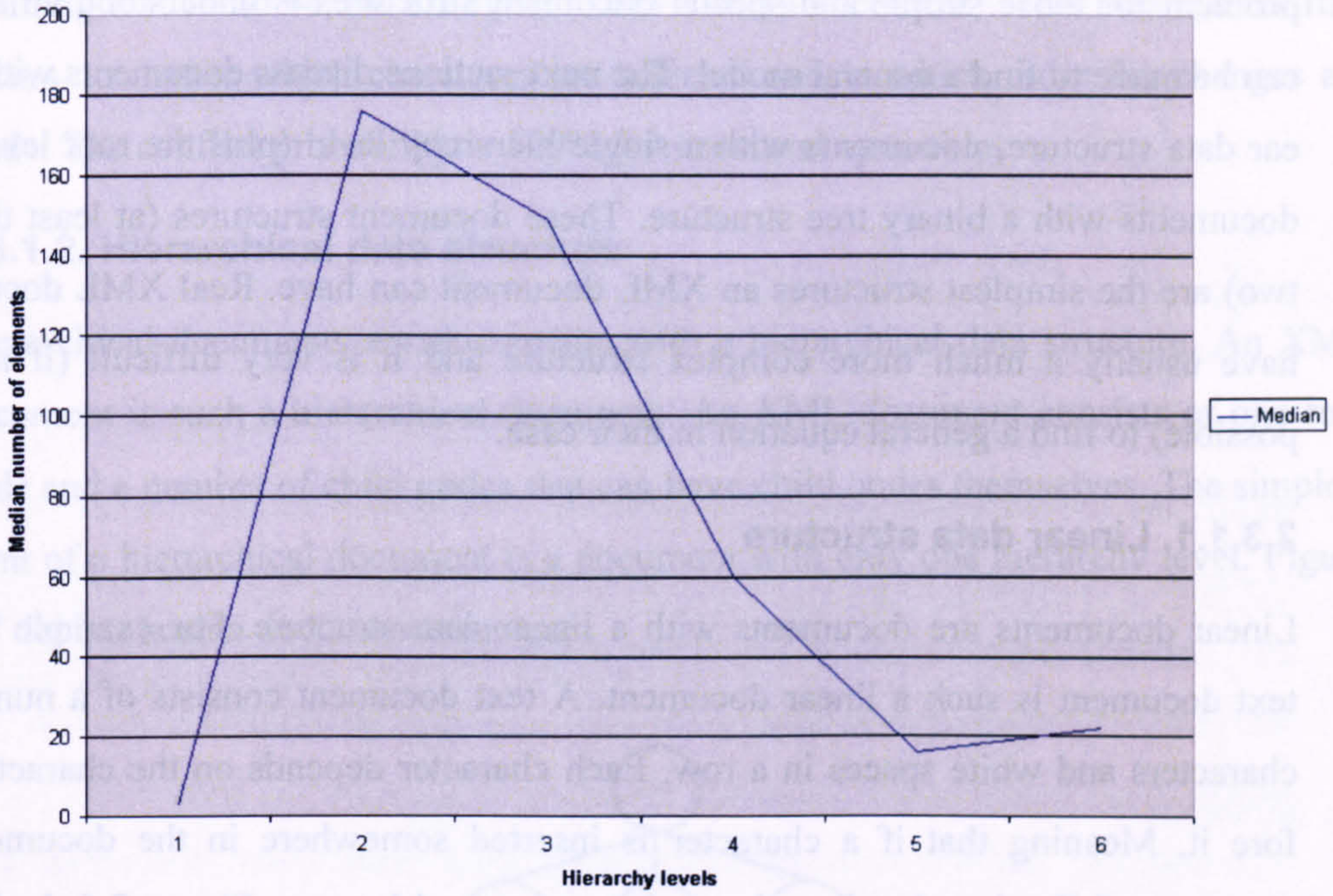

Figure 2.5: Median distribution of elements on hierarchy levels

Based on the analysis results it can be assumed that a typical structure of an SVG document looks roughly like the silhouette of a pear. This structural information is used later in the simulation of conflicts for creating documents (see section 2.3.2).

2.3. Analysis of conflict probability

A theoretical and practical analysis of the conflict probability was conducted. This was done in order to facilitate the development of an optimal concurrency control algorithm for XML documents and to understand how conflicts occur when multiple persons work on one XML document collaboratively in real-time. The practical analysis was conducted by simulation and is discussed in section 2.3.2. The following section discusses the theoretical analysis and the probability of conflicts model.

2.3.1. Theoretical model for the probability of conflicts

A theoretical model can help to better understand the problems in collaborative XML editing. Taking a look at simple and special document structures can help in the development of a theoretical model for calculating the conflict probability. Once the

problem for those simple and special document structures is understood an attempt can be made to find a general model. The next sections discuss documents with a linear data structure, documents with a single hierarchy level (plus the root level) and documents with a binary tree structure. These document structures (at least the first two) are the simplest structures an XML document can have. Real XML documents have usually a much more complex structure and it is very difficult (if not impossible) to find a general equation in their case.

2.3.1.1. Linear data structure

Linear documents are documents with a linear data structure. For example a plain text document is such a linear document. A text document consists of a number of characters and white spaces in a row. Each character depends on the characters before it. Meaning that if a character is inserted somewhere in the document, all characters following that insertion point are moved by one. Figure 2.6 depicts the

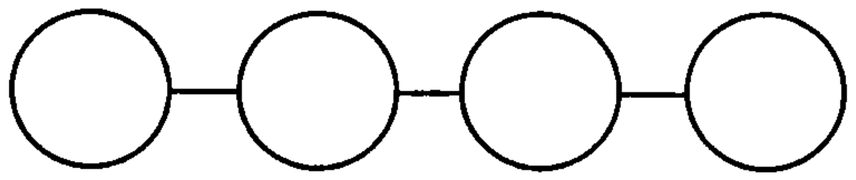

Figure 2.6: Structure of a linear document

structure of such a linear document.b

If two or more people work on a linear document concurrently the probability of conflicts is $P(c) = 1$ in the case of structural changes such as deletion or insertion. To explain this, the following scenario is described: For example the first person selects any character in the document and deletes it. The characters following the deleted character are moved. If at the same time the second person for example inserts any character at any other position in the document, the characters following this insertion position are moved as well and the two operations collide because either the inserted character is placed at the wrong position in the text or the wrong character is deleted. For example, the document contains the text "I am an liner document". The first character is located at position 1 and the last character is located at position 22. If the first person deletes the character at position 7 in order to change the text to "I am a liner document", all characters following it will be moved. At the same time the second person inserts a character at position 12 (between the characters “e” and “r”) in order to change the text to "I am an linear document". If the changes are made the

resulting text could be, for example: "I am a linera document". This means a conflict occurs no matter which characters are selected and in which order the changes are made. The conflict probability is 100% for structural operations.

2.3.1.2. Hierarchical data structure

Hierarchical documents are documents with a hierarchical data structure. An XML document is such a hierarchical document. An XML document consists of one root node and a number of child nodes that can have child nodes themselves. The simplest form of a hierarchical document is a document with only one hierarchy level. Figure 2.7 depicts such a document structure.

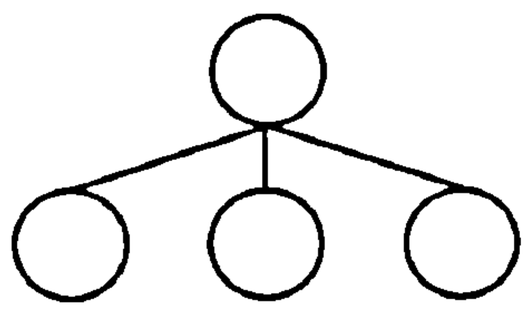

Figure 2.7: Hierarchical document with one hierarchy level

An XML document with this simple structure could look for example like this:

```
<XHTML>
        <HEAD/>
        <BODY/>
        <FOOTER/>
</XHTML>
```

The main difference to a linear document structure is the fact that nodes on the same hierarchy level are not linked to each other. The nodes are linked to their parent node (in this case the root node) and their child nodes (if any). Assuming (for this simplified model) that the order of the nodes within a hierarchy level is not relevant, a deletion of one node or an insertion of another node into the same hierarchy level does not affect the other nodes. Thus a conflict can only occur, if the same node or its parent node is concurrently selected by more than one person and at the same time changed in a way that leads to a different document state at each editing site. A con-

flicting situation exists, for example, if one person deletes a node while another person changes an attribute of the node or inserts a child node. For example, a concurrent insert of a child node and the modification of the same node's attributes does not lead to a conflict.

In this simplified model it is assumed that a conflict occurs if two or more persons select the same node or depending nodes (child or parent nodes) at the same time. Thereby, they change the structure of the document in a conflicting way, for example, by concurrently deleting a node and inserting a child node into it. If, for example, two persons work on the above document concurrently, the conflict probability depends on the number of nodes below the root node. The easiest way to determine the conflict probability here is to constrain the selection of nodes of each person to all nodes below the root node only. In that case the probability for a conflict with two persons working concurrently on a hierarchical document with one hierarchy level is

$$P(c) = \frac{1}{n} \qquad \textbf{Equation 2.1}$$

where n is the number of nodes below the root node. This is valid for conflicting operations and without selecting the root node. If the root node can be selected by any of the two users the conflict probability calculates:

$$P(c_0) = \frac{(n+2(n-1))}{n^2} \qquad \textbf{Equation 2.2}$$

where n is the number of nodes.

2.3.1.3. Binary tree data structure

The more complicated the document structure gets, the more complicated the equation to calculate the conflict probability for structural operations is.

A well known hierarchical structure is the binary tree. In order to find an equation for calculating the conflict probability for this special data structure, different documents were looked at that have a typical binary tree structure such as the examples shown in figure 2.8:

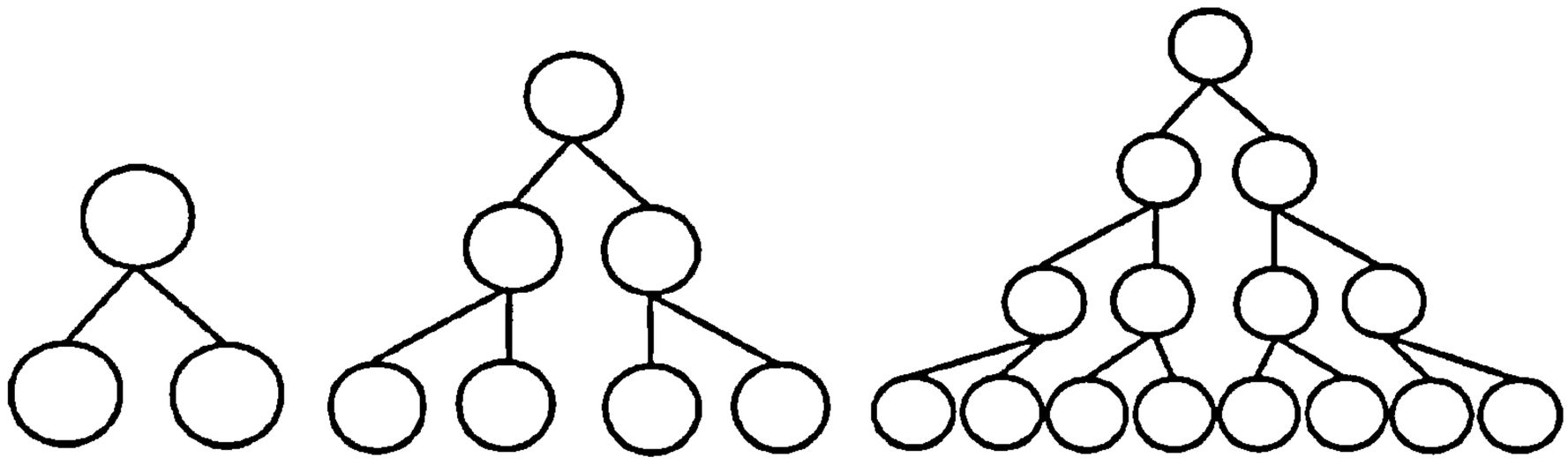

Figure 2.8: Binary tree structures with hierarchy levels 1 to 3

The equation to calculate the conflict probability for two users working concurrently on documents with a binary tree data structure is:

$$P(c_1) = \frac{(n+\sum_{a_i=0}^{i} a_i \cdot 2^{(a_i+2)})}{n^2}$$ **Equation 2.3**

Where n is the number of nodes in the document without the root node and a is the hierarchy level of the document minus 1. The equation was developed by analysis of simple example documents. It is based on the classical definition of discrete probability distributions:

> "Initially the probability of an event to occur was defined as number of cases favourable for the event, over the number of total outcomes possible in an equiprobable sample space."[23]

By counting all possible accesses where a conflict occurs when two users access a document and division of these by the number of all access possibilities, the conflict probability can be calculated. This was done for a few simple documents, the results were analysed and the above equation was developed.

As stated above, the operations that can lead to a conflict are in most cases structural operations that change the structure of a document tree. These are for example delete or move operations. Insert operations do not necessarily lead to a conflict, because the insertion of a node does not influence other nodes in the tree directly. Conflicts only occur if for example one node is deleted while at the same time a new node is inserted beneath the deleted node. Update operations can also lead to a conflict, if for

[23] Probability theory: http://en.wikipedia.org/wiki/Probability_theory retrieved October 30, 2007

example two users change an attribute or content of the same node to different values and the values cannot be merged automatically. To reduce the complexity in this model a conflict occurs if the same set or subset of nodes is selected by the two users concurrently. This means that a conflict for example also occurs if one user moves a node set and the second user updates a node within that node set. In reality this would not necessarily lead to a conflict. The second user would probably be confused, because the changed node suddenly appears at a different location within the document, but the required consistency of the document technically may still be preserved. A question is if the semantic consistency in this case would be preserved as well. An example is shown below for an XML document having a binary tree structure with three hierarchy levels (see figure 2.9) and two users working on it concurrently. The document consists of 14 nodes plus the root node.

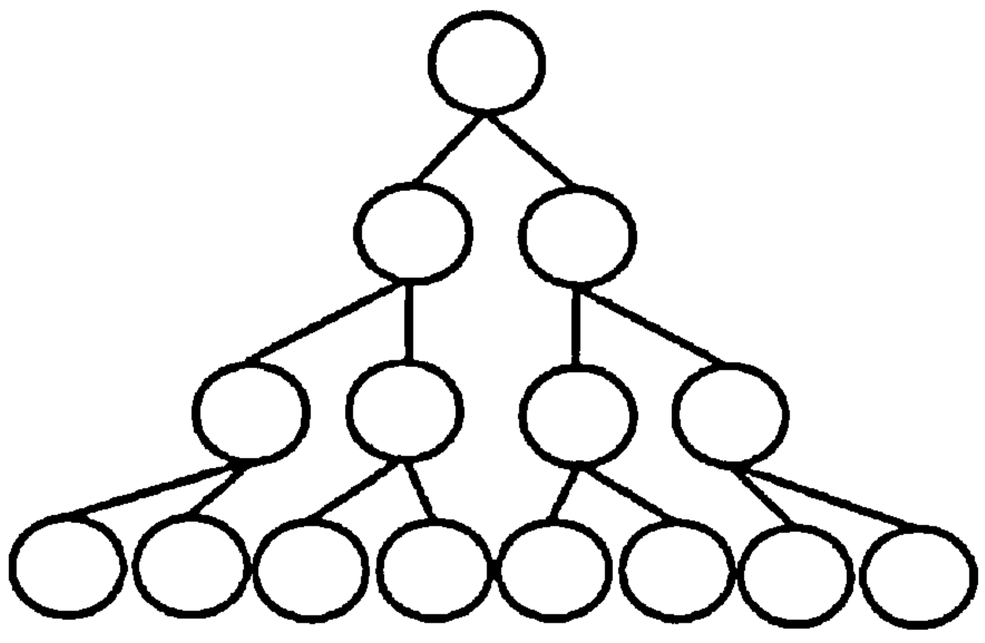

Figure 2.9: Binary tree with three hierarchy levels

The conflict probability is calculated as shown using equation 2.3:

$$P(c_3) = \frac{(14+0\cdot 2^{(0+2)}+1\cdot 2^{(1+2)}+2\cdot 2^{(2+2)})}{14^2} = 0.27551 \qquad \textbf{Equation 2.4}$$

Thus, the calculated conflict probability is $P(c_3) = 27.55\ \%$

The equation derived by theoretical analysis was experimentally evaluated. In order to do this a simulation of two users working on documents with a binary tree structure was conducted. The simulation results then were compared to the theoretical results. The results are discussed in section 2.3.2. The derived equation for calculating the probability of conflicts is only useful for XML documents with a binary tree

structure. Although this equation shows some important properties of the typical behaviour of the conflict probability of hierarchical documents (see section 2.3.2), an attempt was made to find a more general equation. The next section discusses this attempt.

XML data structures in general

In order to find a general equation for the probability of conflicts when editing XML documents in a real-time collaborative editing environment, some assumptions are made:

- All users working on the document behave in a similar way with respect to the selection and execution of operations. Thus the probability for selecting operations and/or nodes to perform those operations on is the same for every user at any given time.
- The probability of selecting any node (including the root node) is always the same for every node. A continuous probability distribution (Laplace[24]) for the node selection exists.
- For simplicity reasons in the first model only two users work concurrently on the document.
- The structure of the XML document on which the users work is random and not part of this simplified model.

In a real system the probability distribution for the selection of nodes could for example be a Gaussian distribution (normal distribution) or a discreet distribution or any other kind of distribution of probability.

Figure 2.10 shows an example for a normal and a discreet probability distribution. On the X-axis are the nodes of the document from 0 to x (each node is assigned a number between 0 and x, where x is the overall number of nodes n-1) and on the Y-axis the probability for selecting each node is given. The probability distribution depends on the structure and type of document. For example if the document is a text

[24] Pierre Simon de Laplace (1812). *Analytical Theory of Probability*, see http://www.maths.tcd.ie/pub/HistMath/People/Laplace/RouseBall/RB_Laplace.html , retrieved October 30, 2007

document such as a book then perhaps the titles of the chapters change rarely while the different paragraphs or sections are changing quite often. Thus the probability for selecting a chapter title node is lower than the probability for selecting a paragraph node.

The probability distribution of a real world example can be much more complex than depicted in figure 2.10. To keep it simple in the first place a continuous probability distribution is assumed. In reality the tree structure of the document can have an influence on the editing process, because it influences the number of nodes that are selected in a move or delete operation. In this very simplified model it is assumed that the structure has no influence on the selection of nodes. In the enhanced model this influence will be taken into consideration.

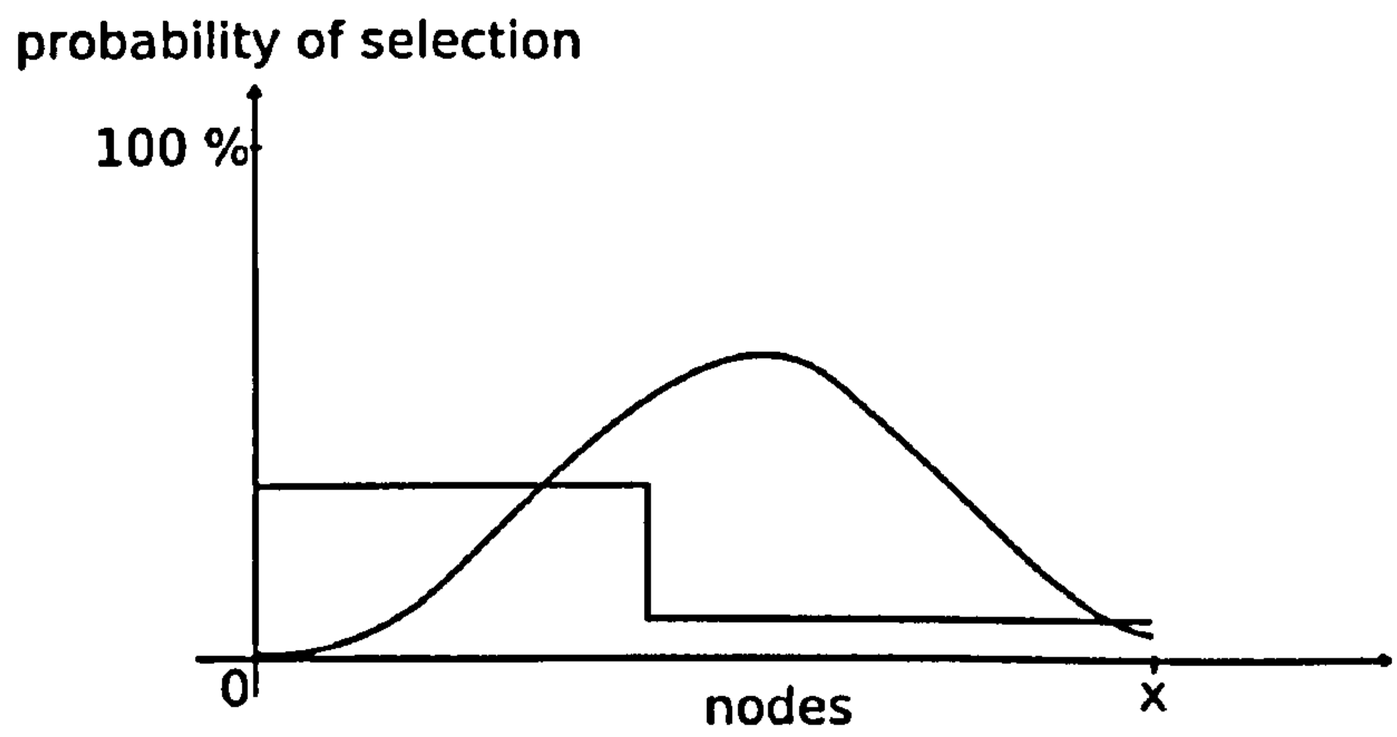

Figure 2.10: Example probability distributions (normal and discreet) for the selection of nodes

The scenario

Two users try to perform operations on the same XML document at the same time. Each user therefore randomly selects an operation at first. An operation can be either a move, delete, insert or an update operation. The operations have the following characteristics:

- Move and delete operations always affect a set of nodes $s >= 1$. If the node on which the operation is to be executed has child nodes, the operation will also af-

fect them. Move and delete operations therefore may require to move or delete a whole sub tree of an XML document

- Insert and update operations always affect one node s=1. It is not possible for one user to insert a node beneath more than one node in only one operation. It is also not possible to update more than one node in a single update operation.

It is assumed that each operation type is selected with the same probability[25]:

$$P = \frac{1}{4}$$ **Equation 2.5**

After the user has selected the operation, a node (element) in the XML document is randomly selected. It is assumed as stated above that each node is selected with the same probability. If a document has n nodes, the probability for selecting a node is:

$$P = \frac{1}{n}$$ **Equation 2.6**

The third step is to execute the operation on the selected node. In order to record conflicting operations, each selected node is marked as "locked" before the operations are executed. If the selected operation was move or delete, the selected node and all its children are marked as "locked". If the selected operation was insert or update, only the selected node is marked as "locked". A conflict occurs if one or more nodes are marked as "locked" more than once. This happens if, for example both users pick the same node, or if one user selects a node that is part of a sub tree in a move or delete operation carried out by another user.

The model

First the probability of conflicts for the operations move and delete is contemplated. With these operation types the number of locked nodes can vary between 1 and n-1 (all nodes except the document root node) for n being the number of nodes in the document. It is assumed that the probability for selecting 1 to n-1 nodes is equal. For example the probability for selecting one node or 10 nodes (by selecting a node with 9 child nodes) is equal:

$$P = \frac{1}{s}$$ **Equation 2.7**

---

[25] The probability for selecting one of the four operation types insert, update, delete and move.

Where s is the number of possible selections.

As stated above the move operation in a real scenario does not necessarily lead to a conflict. In this scenario it is treated as a conflict per definition. This makes the identification of conflicts easier.

If two users edit the same document concurrently they in reality never access a node or a node set at exactly the same time. Accessing a node is done sequentially and this is indispensable due to inherent properties of the used computer architecture. It is done so quickly that the users get the impression of concurrency. For the model this is not relevant, because the order in which the conflicting operations are executed does not have any effect on the fact that the operations collide. It is assumed that the first user accesses a certain number of nodes k and thereby locks them. After that the second user accesses another number of nodes z and tries to lock them as well. A conflict occurs if the second user accessed one or more nodes that have already been locked by the first user. When accessing a document of size (i.e. number of nodes) n, there are

$$\binom{n}{z} . \qquad \textbf{Equation 2.8}$$

different node sets of size z.

The number of combinations where user two does not access at least one of the same nodes as user one is:

$$\binom{n-k}{z} . \qquad \textbf{Equation 2.9}$$

Thus the probability for no conflicts to arise is:

$$P = \frac{\binom{n-k}{z}}{\binom{n}{z}} . \qquad \textbf{Equation 2.10}$$

The probability of conflicts is therefore

$$P = 1-\frac{\binom{n-k}{z}}{\binom{n}{z}} . \qquad \textbf{Equation 2.11}$$

The next step is to take a look at the probability of conflicts for the operations insert and update. In that case the number of nodes that the first user selects is $k=1$. The second user selects $z=1$ nodes. The probability for the second user selecting the same node as the first user is $P=1/n$ where n is the number of nodes in the document. In this case 2.11 reduces to:

$$1-\frac{\binom{n-k}{z}}{\binom{n}{z}}=\frac{1}{n} \quad . \qquad \textbf{Equation 2.12}$$

and

$$\frac{n}{n}-\frac{(n-1)}{n}=\frac{1}{n} \quad . \qquad \textbf{Equation 2.13}$$

The above equation also includes the cases where $k=1$ and $z=n-1$ and the other way round. This occurs if one user selects an insert or update operation and the other user selects a move or delete operation.

Enhanced model

The above equation is correct for the case of two users. If more users are working on the document the improved equation for the probability that no conflicts occur is:

$$P(x_1,x_2,x_3,\ldots,x_L) = \frac{\binom{n-\sum_{i=1}^{L-1} x_i}{x_L}}{\binom{n}{x_L}} \quad . \qquad \textbf{Equation 2.14}$$

L specifies the number of users and x_i is the number of nodes each user has locked. Constraint: the sum of all values of x_i should not be greater than n.

$$\sum_{i=1}^{L-1} x_i \leq n \quad . \qquad \textbf{Constraint 2.1}$$

As stated above it is assumed that the probability distribution for the number of nodes that are marked by each user is a continuous probability distribution (Laplace).

In reality the structure of the document tree has a great influence on the number of selected nodes by each user. Figure 2.11 shows an example document tree.
The probabilities of selecting a certain number of nodes in the shown tree are as follows:

$P(1)=6/10$, $P(2)=0$, $P(3)=2/10$, $P(4)=0$, $P(5)=0$, $P(6)=0$, $P(7)=0$, $P(8)=1/10$, $P(9)=0$, $P(10)=1/10$.

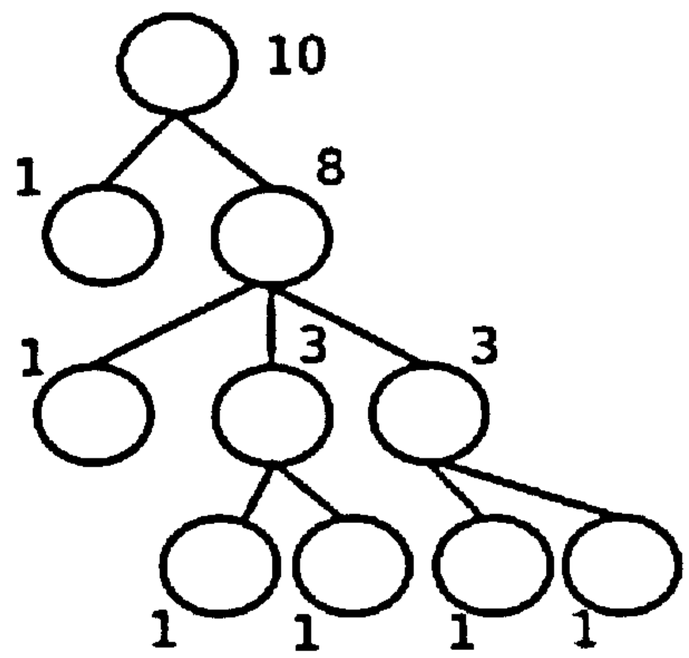

Figure 2.11: Example document tree

To take these probabilities into consideration, the equation is improved in the following way:

$$P(x_1, x_2, x_3, \ldots, x_L)=[\prod_{i=1}^{L-1} P(x_i)]\cdot\left(\frac{\binom{n-\sum_{i=1}^{L-1} x_i}{x_L}}{\binom{n}{x_L}}\right) \qquad \textbf{Equation 2.15}$$

For example three users access an XML document at the same time:

$$P(x_1, x_2, x_3)=P(x_1)\cdot P(x_2)\cdot P(x_3)\cdot\left(\frac{\binom{n-\sum_{i=1}^{L-1} x_i}{x_L}}{\binom{n}{x_L}}\right) \qquad \textbf{Equation 2.16}$$

$P(x_i)$ determines the probability of selecting a certain number x_i of nodes. For example the probability of selecting three nodes in the above tree is

$$P(x=3)=\frac{2}{10} \quad .$$

The probability function $P(x_i)$ depends on the structure of the given XML document tree. Thus the probability of conflicts when more than one users work collaboratively on an XML document is dependent on the structure of the document and the above equation can only be solved correctly for a given document tree.

In this model, the state at a very specific moment in time is looked at. In reality it could happen that, for example, users one and two access the same nodes in the tree but at different times, hence no conflict would occur. This model considers only those moments when two or more users access the document concurrently without virtually any delay. In a more complex model the factor time needs to be taken into consideration. The factor time would change the number of concurrent accesses at a specific time. On a timeline the number of concurrent accesses changes from moment to moment, depending on the number of users and on the duration it takes for each operation to be executed. That is how long a node is locked. The factor time could be expressed in an equation for the probability $P(dt)$ of concurrent accesses within a specific time span. The probability for conflicts within a specific time span is the product of $P(dt)$ and $P(x)$ while $P(x)$ is the probability of conflicts when a certain number of users x access the document of a certain size at the same time. For the probability of concurrent accesses within a specific time span, a Poisson[26] distribution could be applied.

The next step was to verify the equation by means of a simulation. One necessary parameter to calculate the probability was the number of nodes in a document. A set of simple documents was used to simulate conflicts and thus calculate the conflict probability on the basis of the equation. The results of both the calculation and the simulation were compared. The equation produced a contradictory result. In view of the difficulty encountered in deriving this equation it was decided that further work

[26] A discreet probability distribution discovered by Siméon-Denis Poisson (1781–1840). See http://en.wikipedia.org/wiki/Poisson_distribution , retrieved October 30, 2007.

on this was an uneconomical use of time for the anticipated reward. The next section discusses the conducted simulations.

2.3.2. Simulation of conflicts

Simulation is one approach to analyse dynamic systems (one special case is the Monte-Carlo simulation which is used for static simulation). Simulation is used (among other things) if an analysis on the real system would be too complex, too time consuming or the real system does not exist yet. A very common way of simulation today is the computer simulation. In this case the simulation is conducted by using software. Simulation is always based on a model on which experiments are conducted in order to gain knowledge on the system in question. In this work simulation is used to analyse the probability of conflicts when multiple users concurrently work on an XML document. First a dynamic simulation model was developed in order to get a better understanding on how conflicts occur in a collaborative editing process. Next a static simulation was conducted in order to get reproducible results that were used to prove the equations found in the theoretical analysis. The next sections discuss the Monte-Carlo simulation method and the conducted static simulation. The dynamic simulation is discussed in section 2.4.

2.3.2.1. Monte Carlo simulation method

The Monte Carlo simulation is a method for solving various kinds of computational problems by using random numbers (or more often pseudo-random numbers), as opposed to deterministic methods. The Monte Carlo method does not require truly random numbers to be useful. Much of the most useful techniques use deterministic, pseudo-random sequences, making it easy to test and re-run simulations. The only quality usually necessary to perform good simulations is for the pseudo-random sequence to appear "random enough" in a certain sense. That is, that they must either be uniformly distributed or follow another distribution when a large enough number of elements of the sequence are considered. Because of the repetition of algorithms and the large number of calculations involved, Monte Carlo is a method suited to calculation using a computer, utilizing many techniques of computer simulation[27]

[27] Wikipedia: Monte Carlo method. http://en.wikipedia.org/wiki/Monte_Carlo_simulation, retrieved October 30, 2007

(Landau and Binder 2000).

In this work the Monte Carlo simulation method is used for simulating the conflicts occurring in a collaborative XML editing environment. It is used because it is relative easy to formulate and can be applied to nearly all problems. The basic idea is to get a result for the probability of conflicts by repeating the random access of users on XML documents many times and counting the number of conflicts. The found number of conflicts is then divided by the number of iterations. The result is a statistical average number of conflicts, the statistical conflict probability for the analysed setting. The setting in this case consists of certain documents and a certain number of users concurrently working on those documents. The intention of using the Monte Carlo method for simulation was not to find a general equation for the conflict probability when editing XML documents - this general equation is probably very hard to find (if not impossible) - but to get a better understanding of the occurrence of conflicts in a certain setting, which can help in the development of a concurrency control algorithm for XML documents.

2.3.2.2. Static simulation of conflicts

The Monte Carlo method was used to simulate different scenarios. One scenario was: a certain number of users concurrently work on random XML documents. The number of users are 2,4,8 and 16. They work concurrently on documents of different sizes (400-16000 elements) and hierarchy levels (4-10). Another scenario was: two users working on XML documents with a binary tree structure. The intention of simulating the first scenario was to get a better understanding of the conflict probability when working on general XML documents. The intention of simulating the second scenario was to verify the derived equation on the conflict probability when two users work concurrently on an XML document with a binary tree structure. In the following sections the two simulation scenarios are discussed in detail.

Editing general XML documents

In order to analyse the probability of conflicts in a collaborative XML editing environment, an algorithm was developed that simulates a different number of users editing XML documents of different sizes and structures. To get a more realistic simulation result, the XML documents used in the simulation were generated based on

the properties found in the document analysis (see section 2.1).
The simulation software was developed using the Java programming language and the software development environment called Eclipse. The simulation software basically consists of two different parts. The first part is responsible for the generation of the XML documents. The second part is responsible for the overall coordination of the considered tasks, creating the documents, selecting the operations that are to be performed by each "user" and collecting the simulation results. For simulation the number of simulated users, the properties of the documents and an iteration factor are set via a property file. The iteration factor tells how many times the editing process is repeated in the simulation. It is a measure of the accuracy of the simulated results. Increasing the number of iterations leads to smaller error rates in the result. The average error rate is the delta between the simulated results and the real results.
When the simulation is started, the software reads the following parameters:

- Size of the documents (i.e. the number of hierarchy levels)
- Number of users
- Iteration times

A document can have a minimum, a maximum or a fixed number of nodes. For this simulation a fixed number of nodes was used. This means that the number of nodes each document has after it is generated is fix. Once the users have executed operations on the document it is discarded and a new document is generated with that fix number of nodes again. The generated documents had sizes from 400 to 16000 nodes in steps of 400 nodes. That means that first documents were generated with a fixed number of 400 nodes, followed by documents with 800 nodes up to documents with 16000 nodes. The number of hierarchy levels per document were ranging from 4 to 9 levels. These values were based on the document analysis conducted before. The analysis showed that very rarely documents with less than 4 and more than 9 hierarchy levels existed (within the lot of analysed documents).
For each document size (e.g. documents with 400, 800 or 1200 up to 16000 nodes) the operations were performed by first 2 then 4, 8 up to 16 users. At each iteration a document was generated of a certain size, the users performed a random operation on

a random node set of the generated document and after this the conflict events were counted. The whole process was repeated 1000 times for each document size and certain number of users. The nodes were selected randomly with a linear probability of $P=1/n$ where n is the number of nodes. The random number generator of the Java 2 Standard Edition was used for all random numbers (the Random class)[28].

In the simulation, the root node cannot be deleted or moved, because this would lead to a conflict in any case and it is not done in practice. A deletion of the root node would not make sense, because the whole document would be lost. A move of the root node would not make sense either, because there would be no place to move it to. A node cannot be moved below itself or its children. The operations that were performed on a node were: delete, move, insert and modify. The operations were selected randomly with a linear probability of $P=1/4$.

The algorithm used to generate the documents was:

First the overall number of nodes and the number of hierarchy levels was set. Then for each hierarchy level the number of elements (nodes) were calculated. This is done by dividing the number of nodes by the number of hierarchy levels and for each hierarchy level multiplying the result with a weight factor based on the distribution curve found in the document analysis (see section 2.1). The analysis showed that the analysed XML documents had an increased number of nodes in the first 2 to 4 hierarchy levels. The higher the hierarchy level the more the number of nodes decrease.

---

[28] Refer to the Java 2 Standard Edition specification for more information on the underlying random number generating algorithm: http://java.sun.com, retrieved October 30, 2007

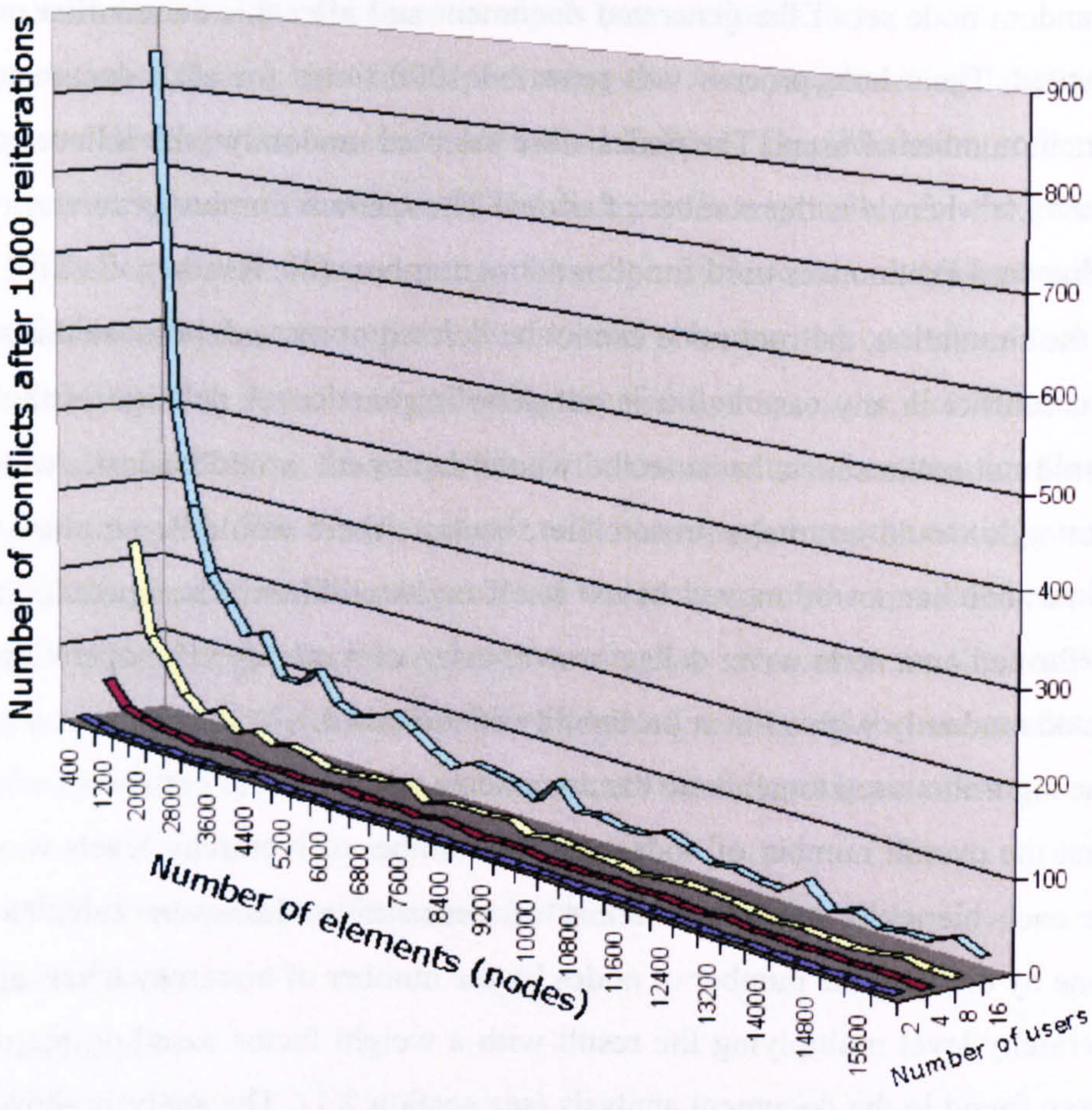

Figure 2.12: Simulation results of the static simulation with number of conflicts per document of a certain size for a certain number of concurrent users

The next step was to create the nodes. Thereafter the nodes were linked to each other and in this way placed into the document. The linking was done by first linking each of the nodes in the first hierarchy level to the root node. After this, each node in the next hierarchy level was linked randomly to another node in the upper hierarchy level. This was done until all nodes on one level were linked to one node in the upper level. Then the nodes in the next lower level were linked to the next higher level. The process was repeated until all nodes in the document were linked. Through this linking method a random document structure was achieved. The simulation results (see figure 2.12) demonstrate quantitatively an inverse non linear relationship between the number of conflicts and the document size. The simulation also confirms that the

number of conflicts grows as the number of concurrent users increases.

The simulation software was later used to demonstrate the validity of equation 2.15 for calculation of the probability of conflicts for general XML documents (see section 2.3.1.3.). The simulation did not use generated XML documents, but existing XML documents instead. First the conflict probability was calculated for certain documents using equation 2.15 and afterwards the simulation was conducted using the same XML documents. Then the results were compared. This was done for a set of different documents and different numbers of users. The results of the simulation and the calculation showed no correlation and it was assumed that the equation found was not correct. The reason for this assumption was the very unlikely calculation results in those cases where the number of users was greater than two. The same method for determining the accuracy of equation 2.15 was used for equation 2.3. Equation 2.3 is used for calculating the conflict probability when editing documents with a binary tree structure. The following section describes the procedure.

Editing binary trees

Again the Monte Carlo simulation method was used for this simulation. A simulation software was written that takes as input an XML document and the number of users that concurrently work on the document. The software then randomly selects a node from the document for each user and checks whether a conflict has occurred. For selecting a node from the document a continuous probability distribution (Laplace) was used. This means that the probability for selecting one node is the same as for selecting any other node in the document. This process of selecting and checking was repeated in every iteration and the number of conflicts occurring was counted. The simulation software was tested with simple documents for which the conflict probability was known, in order to prove the correctness of the simulation results.

As expected the process of randomly picking a node for each user and determining the number of conflicts was more accurate for greater number of iterations. To achieve a practically acceptable result in terms of accuracy, the number of iterations was set to 10^5. In other words nodes were selected randomly 10^5 times from the document for each user and checked for conflicts occurring. In this way the average error rate is reduced. The average error rate in this case is the difference between the

simulated conflict probability and the real conflict probability. Tests with a greater number of iterations (for example one million iterations) have shown no significant reduction of the average error rate, so 100 000 iterations was deemed to be enough to get a sufficiently accurate simulation result.

The results achieved by simulation were compared to the results calculated using equation 2.3. The equation 2.3 to calculate the conflict probability for two users working concurrently on documents with a binary tree data structure is discussed in section 2.2.1.3. Figure 2.13 shows the simulation and calculation results for the first six hierarchy levels of XML documents with a binary tree structure.

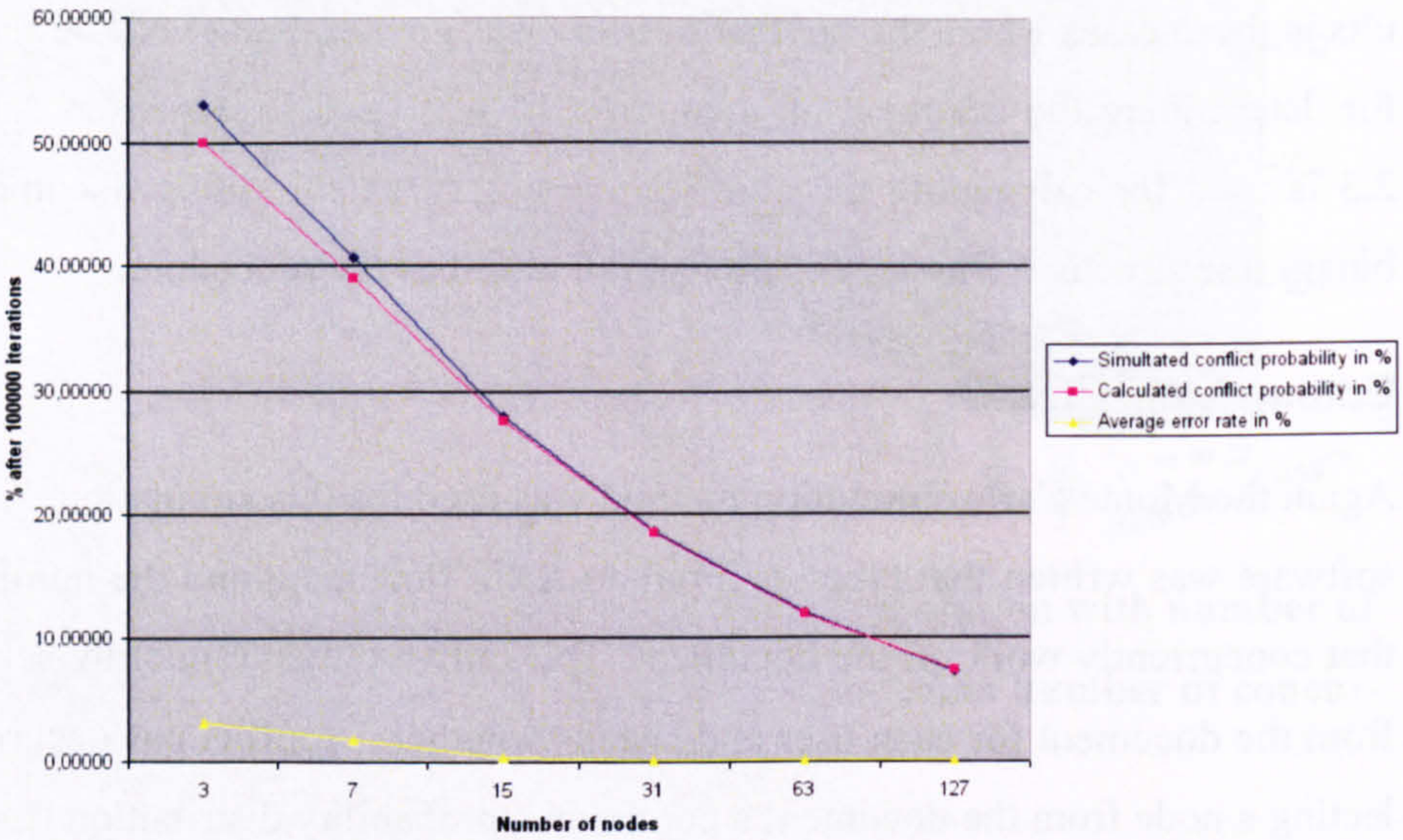

Figure 2.13: Conflict probability for XML documents with a binary tree structure and two users working concurrently

For the calculation and the simulation, the same XML documents were used. The x-axis shows the number of nodes of each XML document and the y-axis shows the conflict probability in percent when editing the documents. Next to the conflict probability, the diagram shows the average error rate for each document.

As the graph shows, the larger (higher number of nodes) the document, the greater the accuracy of the simulation result. The reason for this could be the pseudo-random numbers that are generated by the simulation software. Pseudo-random numbers are not perfectly distributed random numbers. That means they can slightly change the

expected result depending on their quality. The profile of the pseudo-random numbers generated depends on the quality of the pseudo-random number generator and the algorithms it is based on. The pseudo-random number generator used in this case is the standard Java random number generator implemented by Sun Microsystems on the Linux operations system. The greater the range of numbers generated by the pseudo random number generator, the more accurate the Java number generator seems to become. The reason for this could be that the average error of the number generator is split up over the quantity of pseudo-random numbers that are generated. This would explain why, for example, a document with three nodes has an average error rate of approximately 3 % and a document with 31 nodes has only an average error rate of 0,11 %. The above theoretical model for the conflict probability is limited to XML documents with a binary tree structure and two persons collaborating. It is very difficult to find a general model for the conflict probability of all possible XML documents for a variable number of users. But as the above example shows the results that can be achieved by the Monte-Carlo simulation are quite accurate, especially if the documents used are larger than 31 nodes.

2.4. Dynamic simulation of conflicts

In order to get a better impression of the behaviour of conflicts in a collaborative XML editing environment an algorithm was developed for a dynamic editing scenario. It simulated a scenario where a number of users work collaboratively on the modification of an XML document. The users randomly insert, move, delete or update nodes within the document. If a conflict occurs the nodes that are concerned are marked in red.

Figure 2.14 shows a screen shot from the dynamic simulation software.

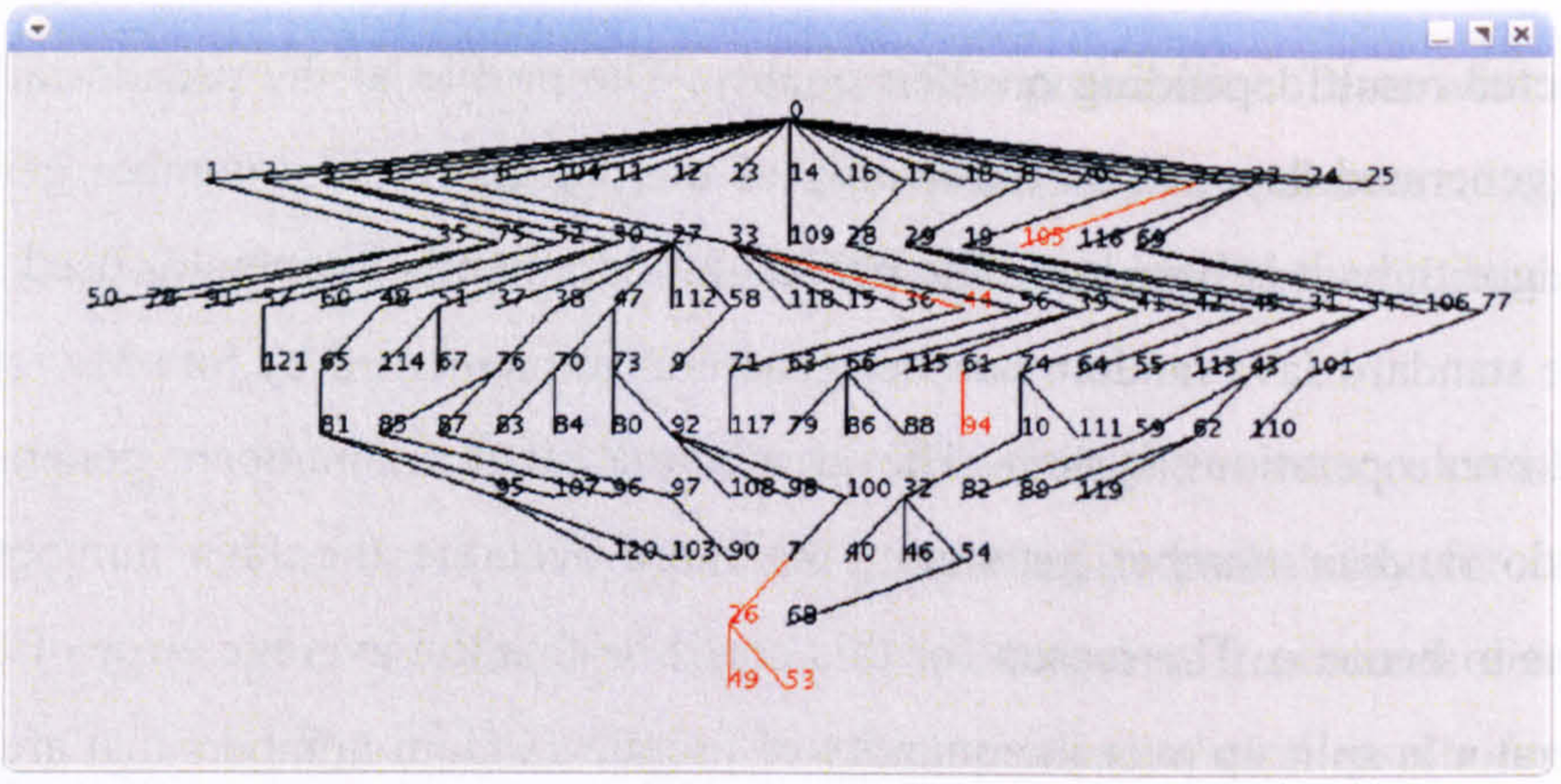

Figure 2.14: Simulation of multiple users working on an XML document concurrently

The simulation algorithm can be broken down into four parts. The first part is responsible for the XML document generation and control. The second part is responsible for the user initialisation and control. The third part is responsible for displaying the document including the changes and conflicts. The fourth part is the so called simulation controller and is responsible for creating the document, the users and the visual components. It starts the simulation, logs the conflicts and tells the visual component to redraw itself if changes occur. The simulation controller reads all properties on the simulation from a property file. The property file contains the following information:

- Number of users (minimum, maximum or fixed)
- Number of elements (minimum, maximum or fixed)
- Number of hierarchy levels (minimum, maximum or fixed)
- Network bandwidth, simulated network speed (minimum, maximum or fixed)
- Delay of operations (minimum, maximum or fixed)
- Allowed type of operations (insert, delete, move, update)
- Different sets of operations
- User delay (minimum, maximum or fixed)
- Name of file for logging

The property "number of users" indicates how many users work collaboratively on the document. If the minimum and maximum values are set, a random number of users between those two values is chosen. If the "fixed number" property is set, the given fixed number of users are simulated. The same way it behaves with all properties where a minimum, maximum or fixed value can be given.

The properties "number of elements" and "number of hierarchy levels" indicate the size of the document that is to be edited. The XML document is created on launch of the simulation based on these properties and the properties found in the document analysis. The algorithm for creating the document is the same as described in section 2.3.2.2. The "network bandwidth" property defines a factor for the simulated network speed. The network speed has an implication for the duration of an operation. A similar implication has the value set for the "delay of operations" property. This property indicates how long an operation may take. The "user delay" property indicates the time between two operations that are performed by a user on the document. All properties can all have a minimum, a maximum and a fixed value, the latter having precedence.

The properties for "allowed type of operations" indicate which operation can be selected and performed by any user on the document. By this it is, for example, possible to only allow insert and move operations but no delete or update operations. To get a more realistic simulation result, a set of operations can be defined. This makes it possible to influence the frequency of certain operations more easily. Usually a user does not, for example, perform insert and delete operations at the same frequency. In most cases a user would, for example, do many inserts before a delete is performed. The last property indicates the name of the log file for registering occurred conflicts.

When the simulation software is started, the simulation controller creates the visual component, called `TreeVisualiser` and reads the properties from the property file. The `TreeVisualiser` component is responsible for drawing the document tree on the screen. Next, the simulation controller generates the XML document on the basis of the properties by calling the `createXMLDocument()` method on the `XMLDocumentCreatorfactory` class. The generation of the XML document works as described in section 2.3.2.1. The next step is to create and initialise the users and the

`ConflictLogger`. The `ConflictLogger` component is responsible for logging any conflicts that occur during the simulated editing process. This is done to provide a record of occurred conflicts. After this is done, the users start editing the document. If a change is made to the document, the `TreeVisualiser` component is notified and then repaints the document tree. If a conflict occurs, the nodes and the links between them are marked in red. Figure 2.15 shows an overview of the different classes (components) of the simulation software and their coherence[29].

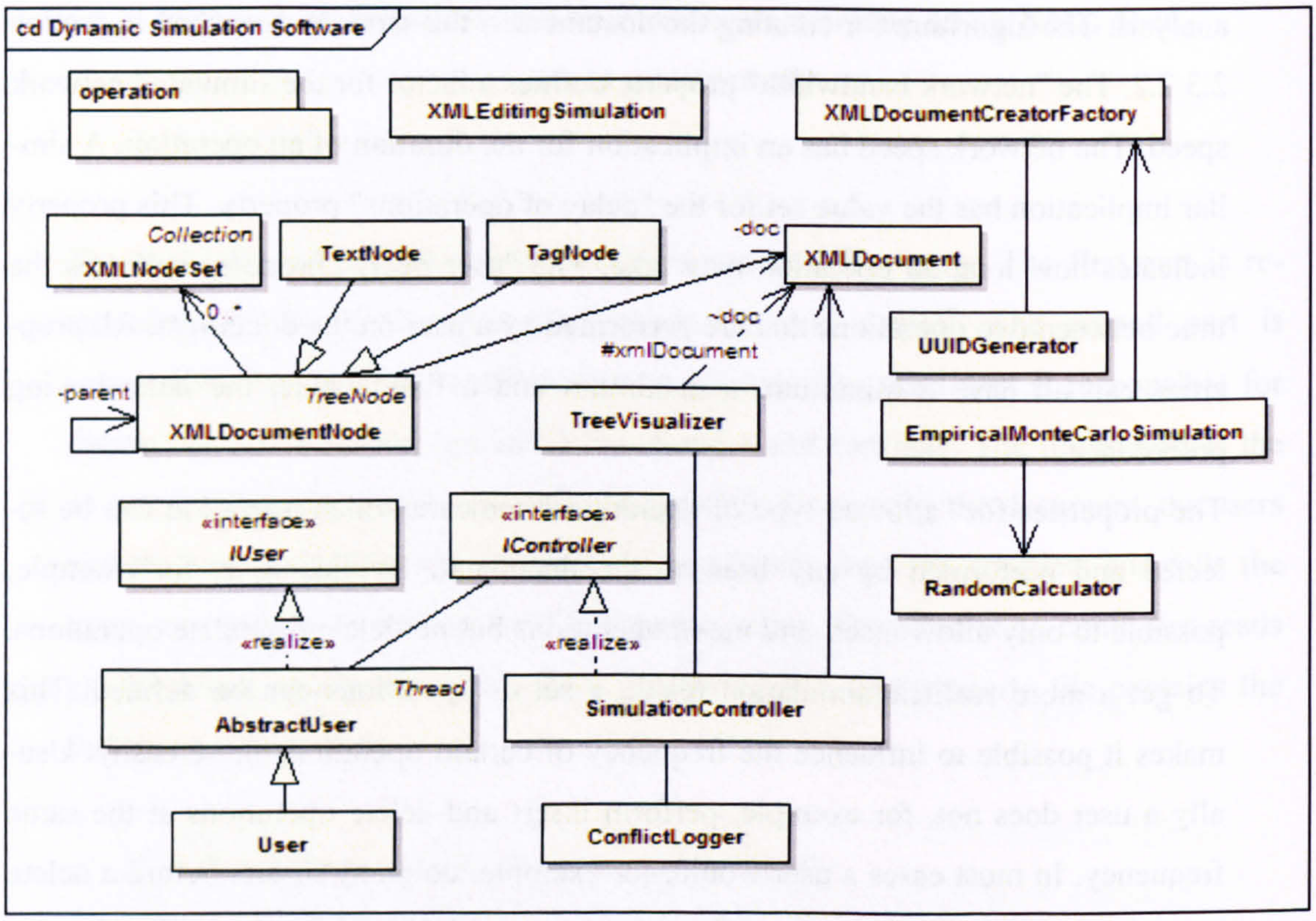

Figure 2.15: UML Diagram on dynamic simulation software classes

2.5. Conclusions

The purpose of the Monte Carlo simulation was firstly to demonstrate the devised equations for the simple binary tree structures were correct. Secondly to simulate concurrent editing on more complex structures in order to gain knowledge of the conflict probability in real concurrent editing situations. The results of section 2.3.2

[29] See appendix for detailed class diagrams

showed how the number of conflicts relate to the increasing size of documents and the number of workers on it. The simulation conducted for a specific set of XML documents with a binary tree data structure confirmed the calculated results for binary tree structures (figure 2.13). The simulation results for arbitrary document structures did not confirm the equation derived for arbitrary XML document structures and was not further investigated. The dynamic simulation algorithm enabled a good understanding of the conflicts to be obtained that occur in collaborative XML editing sessions. The visual component (figure 2.14) showed how the XML document grew over time and the number of conflicts decreased. By changing the parameters it was possible to simulate many different scenarios and observe the affects on the document structure and the conflict frequency. For example, increasing the number of delete operations lead to a shrinking document with increasing conflicts. Decreasing the delete operations and increasing the number of insert operations lead, as expected, to a growing document with reduced number of conflicts. Increasing the number of concurrent users also increased the number of conflicts at the beginning and lead to a rapidly changing document.

The experiences made with the dynamic and the static simulation were very helpful in the later development of the consistency maintenance algorithm for XML documents which is discussed in chapter 3. These results were used as a catalyst for the development of a concurrency control algorithm for XML documents.

Chapter 3. Consistency maintenance in hierarchically structured documents

The core of a collaborative editing framework is the consistency maintenance algorithm. In order to develop an algorithm for the synchronisation of shared XML documents, existing algorithms were analysed. The majority of the existing consistency maintenance algorithms for real-time editing systems today work on a linear data model. However, for this work an algorithm is required that supports the hierarchical structure of XML documents. In recent years different research groups have developed consistency maintenance algorithms for hierarchical data models. Some of their ideas have been adopted and extended in this work in order to develop an algorithm that is specialised on the synchronisation of XML documents. Thus in the following section the most important of the latest research projects are briefly discussed, before a new consistency maintenance algorithm for XML documents is presented.

3.1. Contemporary work

Davis, Sun et al. (2002) propose a model for an operational transformation algorithm working on groves of the Standard Generalized Markup Language (SGML). As XML is a subset of SGML the principles of the algorithm proposed by Davis et al. can be applied to XML documents as well (Davis, Sun et al. 2001). The set of fundamental operations their model supports are insert, delete and update. The nodes in a grove are addressed by a positional addressing scheme where the minimum of nodes have names. The nodes are identified by their position in relation to certain nodes chosen as landmarks. In this way it is not necessary to assign a name or unique identifier to each node within the document, which saves memory space. Their proposed concurrency control algorithm achieves convergence of the documents by operational transformation. Executed operations are kept in a local history buffer. If a remote operation is received it is checked for causal readiness. If the operation is not causally ready, it is not executed until it becomes causally ready. An operation that is causally ready is executed after transforming it (if necessary) by taking into consider-

ation the operations in the history buffer. In order to check operations for causal readiness, each operation is time-stamped with a vector clock before transferring it to the other sites. For intention preservation they introduce the notion of a definition and an execution context. In a single-user editor the definition and execution context of an operation are always identical. In a collaborative system the execution of an operation at a site can have a different context than the context that defined that operation at another site. A set of transformation functions for structural operations are defined based on the GOTO operational transformation control algorithm (Sun and Ellis 1998).

Galli (2000) developed a real-time collaborative editing system for VRML[30] documents. The documents are replicated on each editing site by use of the Mu3D (Galli and Luo 2000) replication protocol. Mu3D update messages are transmitted to inform each site of changes made by other users. For maintaining the consistency of the shared documents a distributed memory consistency system (DMCS) is defined. The abstract data model that is used in this system is based on VRML and Open Inventor[31] nodes. The VRML scene graph (an ordered collection of VRML nodes) is a forest of trees. The single nodes in a VRML tree are addressed by a path starting from the root node. The basic operations supported by the DMCS are read and update (write). All operations are executed locally first. Update operations are sent encapsulated in a Mu3D message to the other sites. To avoid conflicts and achieve convergence, the consistency model, called flow consistency model, locks a node and its child nodes (sub tree) if it is selected by a user. The other sites are then notified of the selection by a control message. The flow consistency model with its locking strategy also preserves causality and users intentions. A disadvantage of this concurrency model is that other users may not work with a locked branch, limiting the options for collaboration, and conflicts are prevented rather than repaired (Davis, Sun et al. 2002).

Ionescu and Marsic (2000) propose an algorithm called dARB for concurrency con-

---

30 VRML – Virtual Reality Modelling Language, ISO/IEC 14772-1:1997 Standard.

31 SGI Open Inventor, object-oriented 3D programming tool kit , http://oss.sgi.com/projects/inventor/, retrieved October 30, 2007

trol in collaborative applications. The abstract data model that is used is a tree. Nodes in the tree are assigned ids. The ids are generated by appending an index number to the site number and the position of the node in the path. For example, for the site with the id = 1, a paragraph can have the ids: 1.1, 1.2, etc., a sentence can have ids: 1.1.1, 1.1.2, etc. The basic operations supported are: create, delete and edit properties. Each operation is executed locally first which ensures good responsiveness. After that an event is generated that contains the following information: the id of the node, the path from root to the new node and the operation itself. If the operation is "edit property" additionally the information about the property changes are contained in the event. The event is then transferred (via multicast) to the other sites. The algorithm uses a partial ordering based on a vector clock to check if concurrent operations overlap and access the same node. If no overlap exists the operation is executed. Otherwise a so called arbitration phase is started in order to resolve the conflict. Within this arbitration phase each site sends a special event with a computed priority to the other sites. The site with the highest priority wins, all other conflicting operations are discarded and the operation of the winning site is executed at all sites. The disadvantage of the arbitration method is that the intentions of the losing sites are lost and an overhead is caused by running a distributed arbitration algorithm every time a conflict occurs.

Molli, Skaf-Molli et al. (2002) developed a collaborative system that allows editing of simple HTML documents and CRC[32] cards in a synchronous, asynchronous or multi-synchronous mode. A user can switch between the different modes during the editing process. In the asynchronous mode each operation is executed locally and stored in local log. If the mode is switched to synchronous, the operations are propagated to the other sites. Before the operations can be propagated the concurrent operations need to be integrated first. In the integration phase conflicts are solved by the underlying concurrency control algorithm. The multi-synchronous mode works similar to version control systems such as the Concurrent Versions System (CVS)[33],

[32] Class-Responsibility-Collaboration cards (CRC cards) are a brainstorming tool used in the design of object-oriented software

[33] Concurrent Versions System, open-source version control system, http://www.nongnu.org/cvs/, retrieved October 30, 2007

ClearCase[34] or Subversion[35]. Each site maintains a local copy of the shared data. At certain times the copies are merged and if conflicts exist, they are (if possible) resolved. The abstract data model that is used is a tree. The example application uses an XML document to represent the CRC cards and the HTML document. The set of operations supported are: creation of a node, deletion of a node, creation of an attribute, deletion of an attribute and change of an attribute. For concurrency control the SOCT4 (Vidot, Cart et al. 2000) operational transformation algorithm is used. Operational transformation is used here for mutational operations, not for structural operations. The consistency model applied is the same as in Sun and Chen (2002).

Ignat and Norrie (2002) propose a tree-based model algorithm called treeOPT. The algorithm relies on operational transformation and is based on the same consistency model as described by Davis, Sun et al. (2002). The abstract data model is a tree of nodes. In contrast to Davis, Sun et al. a history buffer is kept for each hierarchy level (path) in the tree instead of keeping one for the whole document. This has the advantage that operations that are executed on a certain hierarchy level do not need to be checked against all concurrent operations that have already been executed on the document. The new remote operation only needs to be checked against the operations in the history buffer of the concerning hierarchy level. This saves processing time and can lead to a better performance. The abstract data model used in their sample text editor application has a fixed number of hierarchy levels: 0 for document level, 1 for paragraph, 2 for sentence, 3 for word and 4 for character level. A node is addressed by a path assembled from vectorial positions for each hierarchy level. The vectorial positions specify the positions for the levels corresponding to a coarser or equal granularity than the granularity of the operation. For example, an insertion operation on word level (3) has to specify the paragraph and the sentence in which the word is located, as well as the position of the word inside the sentence (Ignat and Norrie, 2002).

---

[34] Rational Clear Case, software tool for revision control developed by Rational Software, http://www-306.ibm.com/software/awdtools/clearcase/, retrieved October 30, 2007

[35] Subversion, open-source software for revision control, http://subversion.tigris.org/, retrieved October 30, 2007

3.2. A new algorithm for synchronisation of XML documents

Following a definition of Sun and Chen (2002) a cooperative editing system is said to be consistent if it always maintains the properties of convergence, causality preservation and intention preservation (see chapter 1.). The same principles are adopted here, with the exception that operational transformation for solving the intention preservation problem is not used. A new method to preserve intentions of structural operations similar to the approach of Molli, Skaf-Molli et al. (2002) is applied. In contrast to Molli, Skaf-Molli et al., the operational transformation for update or mutational operations respectively is not used. As shown in chapter 2 the probability for a concurrent modification of the same attribute of a node is relatively small when editing an average sized or large document. Therefore it was decided to let the users solve the problem instead of combining attribute values by operational transformation. Another reason for this is that the algorithm is intended to be as general as possible in order to support a diversity of applications. In a professional environment a combination of attribute values to a new, possibly invalid, value would rather be obstructive than beneficial. However, the Collaborative Editing Framework for XML was designed in a way so that modifications to the consistency maintenance algorithm can be easily integrated in order to adapt it to the corresponding application's requirements. In this section, first the above mentioned consistency properties are explained briefly. After that follows an explanation of how these properties are maintained with the new consistency maintenance algorithm for XML documents called CMAX.

Convergence

The convergence property guarantees that when the same set of operations is executed at all sites, all copies of the shared document are identical.

Causality

The causality preservation property requires that operations that are causally interrelated are executed in the right causal order at all sites. Due to Lamport (1978) this causal relation between operations is defined as the causal ordering relation "→". For a definition of the causal ordering relation see chapter 1.

Intention Preservation

The intention preservation property requires that for any operation *O*, the effects of executing *O* at all sites are the same as the intention of *O* when it was generated, and the effect of executing *O* does not change the effects of concurrent (independent) operations.

To satisfy the causality preservation property, a time stamping scheme based on a so called State Vector (SV) is used as described by Sun and Chen (2002). The state vector is a data structure containing a counter for the number of operations executed at each site. Each site maintains a state vector. Local operations are executed immediately. After a local operation is executed, the local state vector is updated and a copy of the current state vector is propagated together with the operation to the other sites. When a remote operation arrives, its state vector is used to compute if the operation is causally ready to be executed. In order to preserve the causality property, operations at a certain site are only executed if they are causally ready. Otherwise the operations are queued until they become causally ready for execution. After the execution of a remote operation at a site, the local state vector is updated correspondingly. The conditions for the causal readiness of a remote operation are defined as follows:

Definition 1: *conditions for executing remote operations.* Let *O* be an operation generated at site *i* and timestamped by SV_o . *O* is causally ready for execution at site *j* ($j \neq i$) with a state vector SV_j only if the following conditions are satisfied:

(1) $SV_o[i] = SV_j[i] + 1$ and

(2) $SV_o[s] \leq SV_j[s]$, for all s $\{0, 1, \ldots, n\text{-}1\}$ and $s \neq i$.

The first condition ensures that *O* must be the next operation in sequence from site *i*, so no operations originated at site *i* have been missed by site *j*. The second condition ensures that all operations originated at other sites and executed at site *i* before the generation of *O* must have been executed at site *j*. Altogether these two conditions ensure that all operations which causally precede *O* have been executed at site *j*. It

can be shown that if a remote operation is executed only when it satisfies the above two conditions, then all operations will be executed in their causal orders, thus achieving the causality preservation property of the consistency model (Sun, Yang et al. 1996).

Each site maintains a device called a history buffer (HB) to store all executed operations. The convergence property is satisfied by ordering all operations in the history buffer based on the total ordering relation "⇒ " and applying an *undo/do/redo* scheme. The total ordering relation is defined as follows:

Definition 2: *Total Ordering Relation* "⇒ ". Given two operations O_a and O_b, generated at sites i and j and timestamped by SV_{Oa} and SV_{Ob}, respectively, then $O_a \Rightarrow O_b$, if and only if (1) $sum(SV_{Oa}) < sum(SV_{Ob})$ or (2) $i < j$ when $sum(SV_{Oa}) = sum(SV_{Ob})$, where $sum(SV) = \sum_{i=0}^{N-1} SV[i]$.

It can be shown that the total ordering relation "⇒ " is consistent with the causal ordering relation "→" in the sense that if $O_a \rightarrow O_b$, then $O_a \Rightarrow O_b$ (Sun, Yang et al. 1996).

When a new remote operation O_{new} arrives at a site and is causally ready for execution, all operations in the history buffer are ordered with respect to the total ordering relation. Then the *undo/do/redo* scheme is applied. First all operations O_b that totally follow new operation O_{new}, that is $O_{new} \Rightarrow O_b$, are undone. In this way the state of the local document is restored to the state before the operations O_b were executed. In the "do" step, the operation O_{new} is executed. In the "redo" step, the operations O_b are redone so that their effect is integrated into the document again. This *undo/do/redo* scheme is executed in the background and the intermediate document states are not displayed to the user. The user only sees the final result of the operations execution. It is worth noticing that this method requires that operations and their executional effect is reversible.

3.2.1. Intention preservation by operational transformation

Applying the above mentioned methods for maintaining the causality and convergence properties can lead to a situation where the intentions of a user are lost. To preserve the intention of operations, Sun et al. propose the transformation of causally ready operations before their execution, to compensate the changes made to the document state by other executed operations. To explain this, consider a concurrent editing session where user one executes the operation O_a at site i $(i=0)$ and at the same time user two executes the operation O_b at site j $(j=1)$. Figure 3.1 shows the two local copies of a shared document before and after the local execution of the operations, including the sites state vector *(SV[...])* and history buffer *(HB[...])* values. The intention of user one is to insert a new node containing the letter *"X"* left of the node containing the letter *"D"*. The intention of user two is to insert a new node containing the letter *"Z"* below the node containing the letter *"D"*.

In order to perform an operation on a node, it has to be identified somehow. The algorithms discussed in Davis, Sun et al. (2002), Ignat and Norrie (2002), Molli, Skaf-Molli et al. (2002), Ionescu and Marsic (2000) and Galli (2000) use a positional addressing scheme based on a path or a relative id for the identification of nodes within a document.

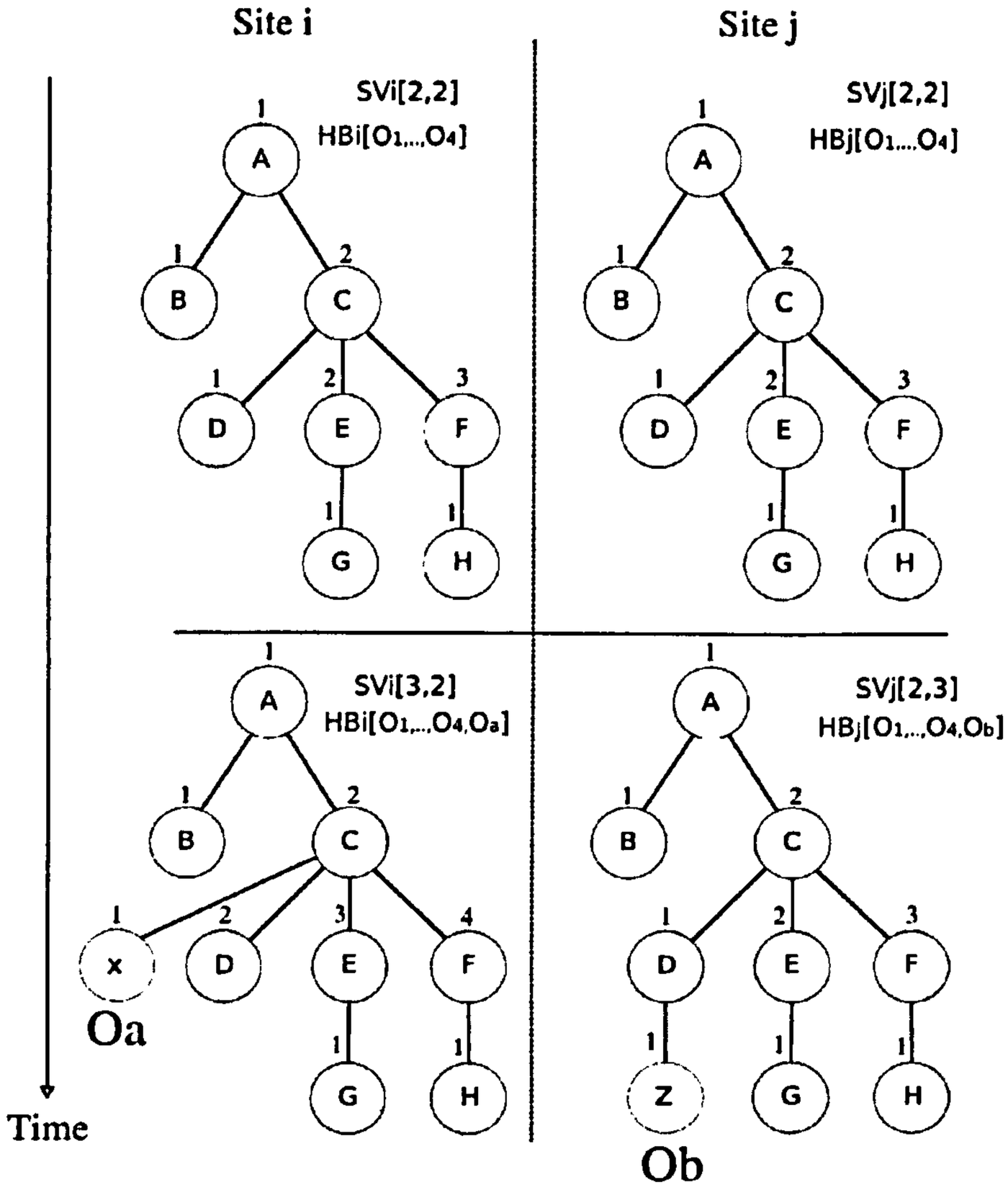

Figure 3.1: An example concurrent editing scenario

In this example the node containing the letter *"D"* is identified by the path *p*="1,2,1" before the execution of O_a at site *i*. Operations may change the relative position of a node within a document and thereby change the path to that node. Thus, the same node containing *"D"* is identified by the path *p*="1,2,2" after the execution of O_a at site *i*. The operation O_a is defined as *INSERT(NODE("X"),p="1,2,1")*. The operation O_b is defined as *INSERT(NODE("Z"),p="1,2,1,1")*. In the first parameter of the operation a new node is created containing the given letter. The second parameter defines the path where to insert this new node. The operations are executed locally first, stored in the local history buffer and then each operation, including a copy of the local state vector *SV[i,j]*, is propagated to the other site. The local state vector at site *i* after the local execution of O_a is $SV_i[3,2]$. The local state vector at site *j* after the local execution of O_b is $SV_j[2,3]$. The history buffer of site *i* after the local execution of O_a contains $HB_i[O_1,...,O_4,O_a]$. The history buffer of site *j* after the local

execution of O_b contains $HB_j[O_1,...,O_4,O_b]$.

When the remote operation O_b is received at site i, it is checked for causal readiness by comparing the state vector SV_{ob} of O_b with the local state vector SV_i at site i. The comparison

$SV_{ob}[j] = SV_i[j]+1$ and $SV_{ob}[i] \leq SV_i[i]$

meets the conditions for executing remote operations and thus O_b is causally ready for execution at site i. The same is true for operation O_a when it is received at site j. The new operation at each site is now ordered with the operations in the history buffer. This is done by comparing the state vector of the remote operation to each state vector of the operations stored in the history buffer. Following the total ordering definition,

if $sum(SV_{Oa}) = sum(SV_{Ob})$ and $i < j$ then $O_a \Rightarrow O_b$.

Assuming that the operations O_1 to O_4 in the history buffer totally precede the operations O_a and O_b, and $O_a \Rightarrow O_b$, the new values of the history buffers at both sites are now $HB[O_1,...,O_4,O_a,O_b]$.

Thus, at site i no operation has to be undone because all operations already are in the right order and O_b is executed directly. At site j, the operation O_b has to be undone, then O_a has to be executed and O_b has to be redone again. The following figure shows the resulting documents at each site after executing all operations.

The documents at both sites are now convergent. However, the intention of user two was to insert the node with the value *"Z"* below the node with the value *"D"*. Now the node *"Z"* is placed below the node *"X"* and the intention of user two is not preserved. The reason for this is explained next.

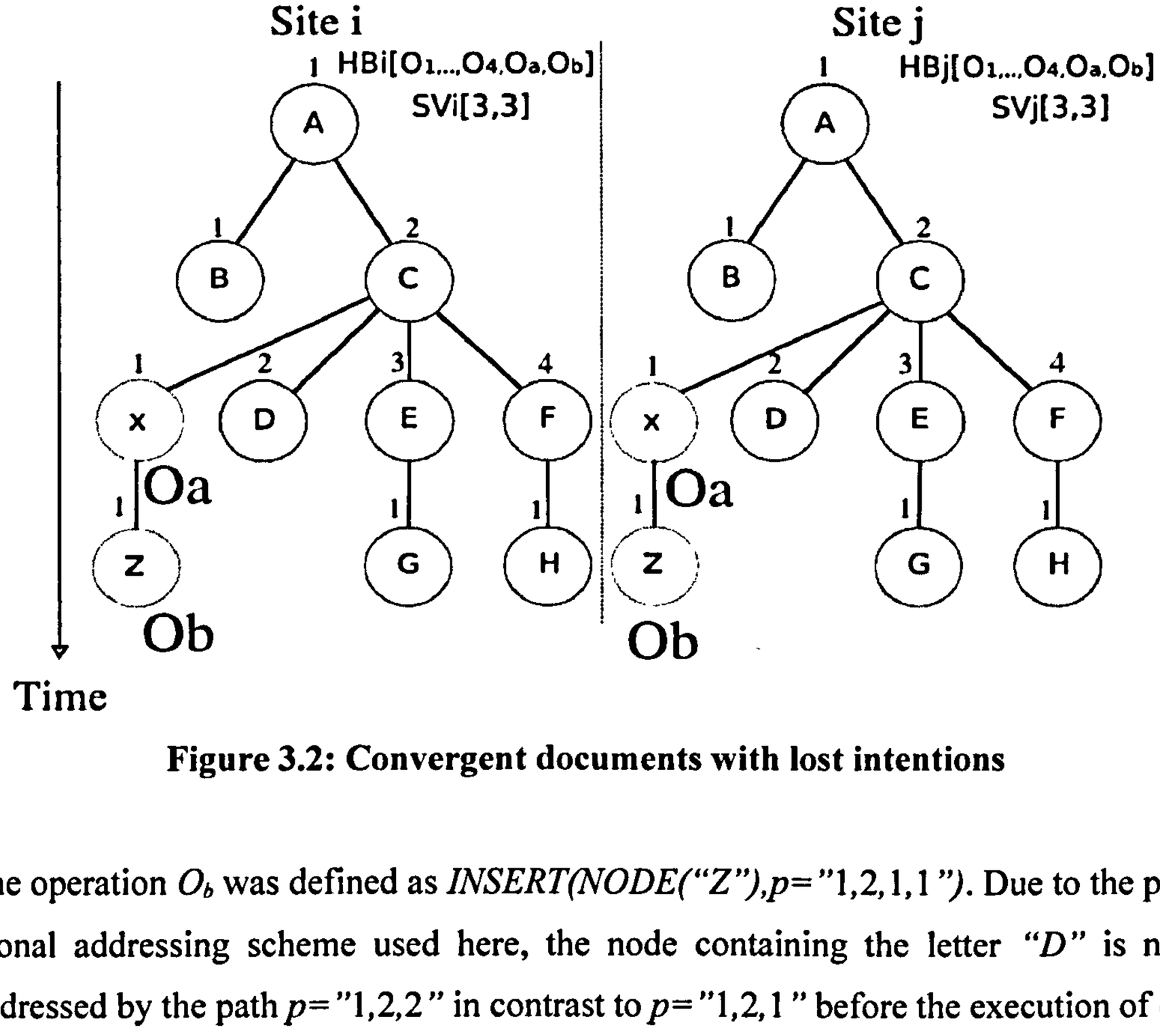

Figure 3.2: Convergent documents with lost intentions

The operation O_b was defined as *INSERT(NODE("Z"),p="*1,2,1,1*")*. Due to the positional addressing scheme used here, the node containing the letter *"D"* is now addressed by the path p="1,2,2" in contrast to p="1,2,1" before the execution of O_a. In order to preserve the intentions of user two, the operation O_b must be transformed so that the impact of O_a is effectively included in O_b. A number of algorithms for operational transformation exist (Davis, Sun et al. (2002), Ignat and Norrie (2002), Molli, Skaf-Molli et al. (2002), Ionescu and Marsic (2000),Galli and Luo (2000), Galli (2000)) that try to solve this problem for hierarchical data structures. In our example the position parameters of the operations O_a and O_b are compared and the resolution is quite simple. When transforming the operation O_b into the form *INSERT(NODE("Z"),"*1,2,2,1*")*, the intentions of user two are preserved and the intention preservation property is satisfied. The resulting final document is shown in figure 3.3.

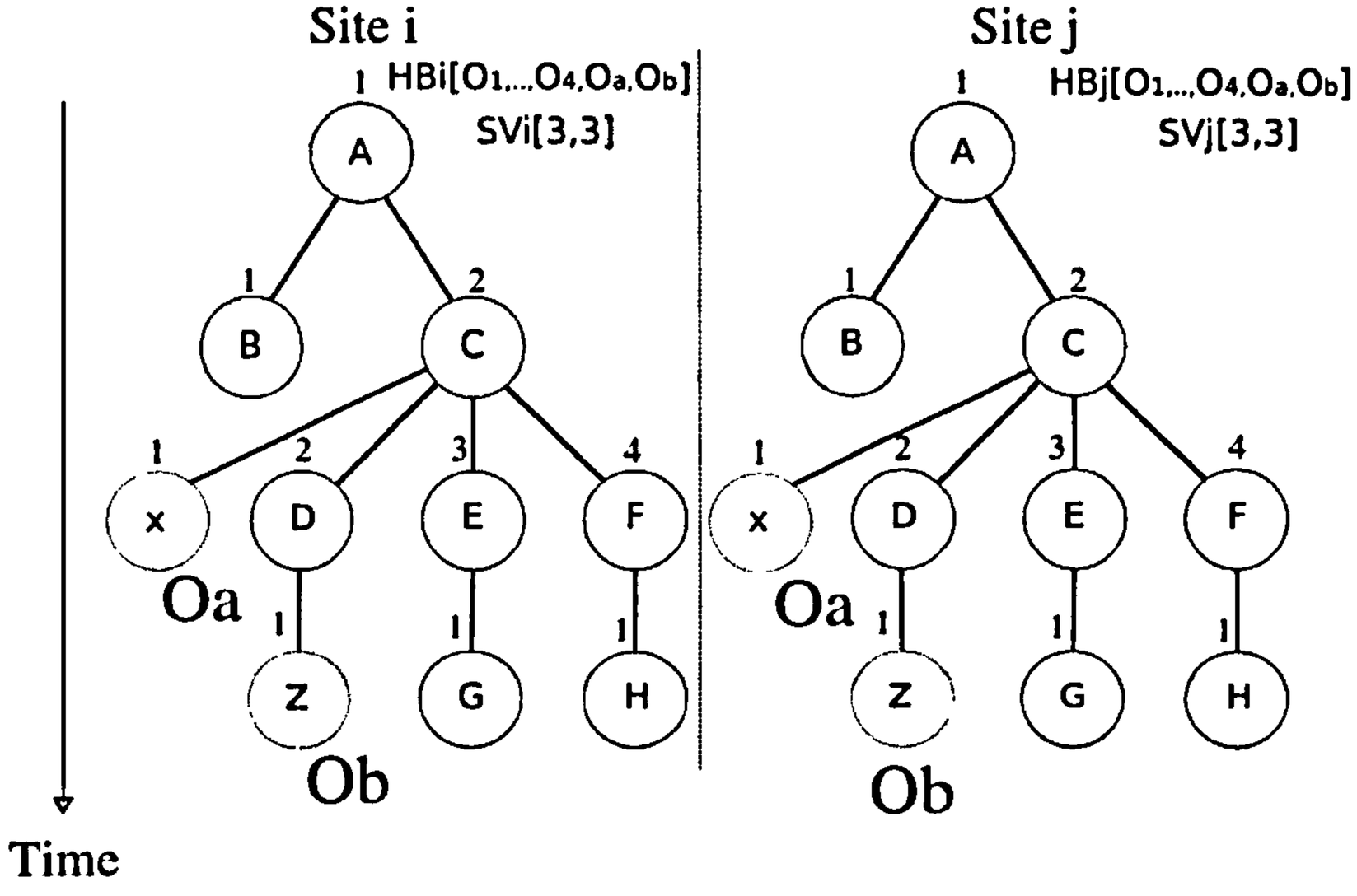

Figure 3.3: Satisfying the intention preservation property by operational transformation

3.2.2. Definition and execution context

The above solution to the problem is based on the fact that the two concurrent (independent) operations were generated on the same document state at each site (for a definition of concurrent operations see chapter 1.5.2.2). However, in an unconstrained cooperative editing environment, it is possible that some concurrent operations are generated from different document states. This can lead to the problem that a simple comparison of the positional parameter of concurrent operations is not sufficient in order to compute the necessary path transformations. To overcome this problem Davis, Sun et al. introduce the notion of definition and execution context (Sun and Ellis1998) and propose a set of inclusion and exclusion transformations for a hierarchical document structure (Davis, Sun et al. 2002).

The context of a document state is defined by the sequence of operations that were executed to bring it from the initial state to the current state. The definition context of an operation *O* describes the document state that existed when *O* was initially created by the user. The execution context of an operation *O* describes the state of the document when *O* is executed on it. In a single user application the definition and the

execution context are always identical. In a concurrent editing application this is only true at the site that generated *O*. In order to effectively preserve the intended effect of a user at all sites, it has to be checked if the definition context matches the execution context at a site. If the definition context of an operation does not include all concurrent operations, for example due to a transmission delay at the generating site, the operation has to be transformed at the receiving site so that the execution context of the receiving site matches the definition context of the generating site. This is done by applying a set of inclusion and exclusion transformations. The inclusion transformations include the effect of an operation as shown in the above example. The exclusion transformations exclude the effect of an operation. This inclusion and exclusion transformation scheme can become very complex when the number of sites and concurrent operations increase. The transformations repeatedly need to be applied to a number of operations which increases computing time and thus reduces performance. In order to achieve intention preservation *and* convergence the *undo/do/redo* scheme needs to be adapted to a *undo/transform-do/transform-redo* scheme, so transformation is applied not only when executing a new operation, but also when redoing operations from the history buffer.

3.2.3. Preserving intentions without OT

The CMAX algorithms preserve the users intentions without using Operational Transformation. As stated above CMAX follows the same principles for causality and convergence as the algorithms described by Ellis and Gibbs (1989). In contrast to the discussed systems CMAX does not use a positional addressing scheme such as paths or relative ids. Instead, each node within a document is assigned a universal unique identifier (UUID). Using a UUID based addressing scheme and identifying a node within a shared document independent of the document structure has advantages over positional addressing.

Consider the same example document as shown in figure 3.1. This time, each node is assigned a unique identifier. The node containing the letter *"D"* now is identified by, for example, *UUID="ABCD"* (for simplicity reasons no real UUID is used here). The node containing the letter *"C"* is identified by *UUID="ABC"*. The operation O_a is now expressed as:

INSERT(NODE("X"),LOC("ABC",REL("ABCD",P=0))).

The operation O_b is expressed as:

INSERT(NODE("Z"),LOC("ABCD",REL("",P=1))).

The first parameter again is a reference to the new node that is to be inserted. The second parameter, *LOC,* identifies the location where to insert the new node. The location is defined by two parameters:

1. The UUID of the parent node
2. The relative position *REL* to a child node of the given parent node.

REL is defined by the UUID of the "checkpoint" and a relative position parameter *P*. The checkpoint is a selected child node of the given parent node. The relative position parameter specifies if the new node will be inserted before or after the checkpoint. The value P=0 indicates that the new node is inserted before the checkpoint. P=1 indicates that the new node is inserted after the checkpoint.

In this example the checkpoint UUID in O_b is empty because the given parent node does not contain any child nodes. When no checkpoint UUID is given, the new node is appended as last child to the parent node when P=1 and as the first child when P=0. However, when a parent node has no child nodes the value of the parameter P is not relevant as a child node can only be appended. As the intention of user two is to insert a new node below the given parent node, appending the new node as last child satisfies this intention. The intention of user one is to insert a new node left of the node containing the letter *"D"*. The location parameter of O_a defines that the node is inserted as child of the node with the *UUID="ABC"* and before the node with the *UUID="ABCD"*.

After executing the operations locally at each site, they are again transmitted to the other site and checked for causal readiness. Then the operations are ordered with the operations in the history buffer at each site. The resulting history buffer values are the same as in the above example. At site *i* the operations are in the right order, so O_b is executed directly. At site *j* the operation O_b is undone, O_a is executed and O_b is redone. The resulting final document states at both sites are convergent and both users intentions are preserved as shown in figure 3.4. By using this new method of addressing a location within a document, it is not required to transform operations in order to satisfy the intention preservation property.

Assigning a unique identifier to each node, on the other hand, requires more space in memory, especially if the shared document contains many nodes. However, the space required for an UUID is relatively small in comparison to the space required for a document. Today, insufficient memory is not a practical problem any more considering the technology developments in computer memory production in recent years.

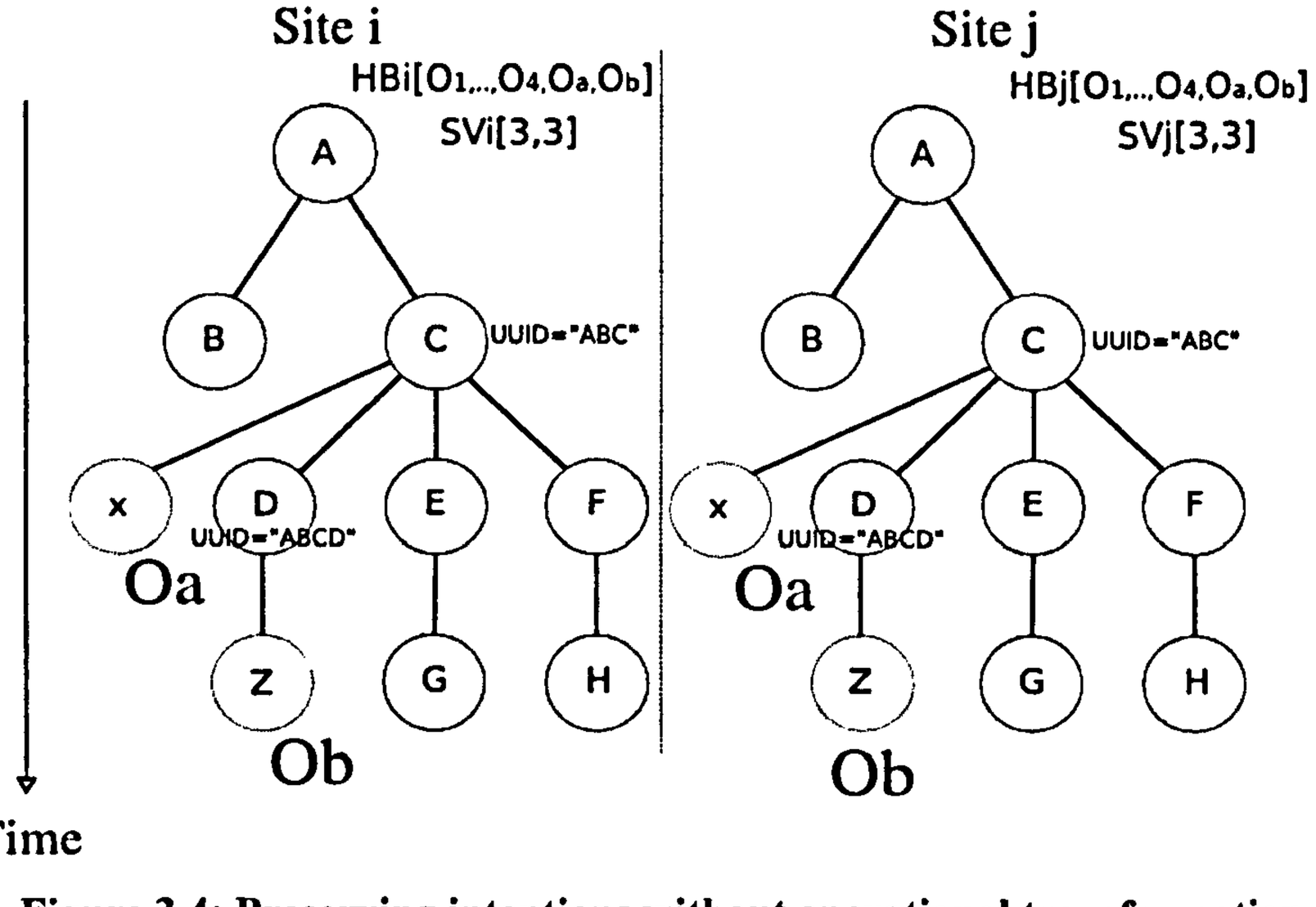

Figure 3.4: Preserving intentions without operational transformation

3.2.4. Conflicting structural operations

Within a real-time collaborative editing session conflicts occur when concurrent operations target the same objects and modify them in a way such that the result is not consistent. It is important to differentiate between the two types of operational conflicts: structural and mutational conflicts. This section discusses how CMAX resolves structural conflicts and maintains the properties of causality, consistency and intention preservation in such cases.

The basic set of operations used are insert, delete and update. Update operations do not modify the structure of a document. They are used in order to change the text contents or the attributes of an XML node. The delete operation removes a certain node from the document, but keeps it memorized in order to be able to undo a delete. The insert operation – as shown above – inserts a node at a certain location within

the document.

The set of conflicting structural operations includes:

1. Concurrent inserts of different nodes to the same location in the document.
2. Concurrent insert and delete operations that target the same node.

3.2.4.1. Conflicting insert operations

There are two kinds of conflicting insert operations. The first kind consists of operations where the target node, that is the node where the new node will be inserted, does not contain child nodes. The second kind are insert operations, where the target node carries child nodes. In the first case the concurrent operations O_a and O_b, for example, are defined as:

O_a = *INSERT(NODE("Z"),LOC("ABCD",REL("",P=1)))*

O_b = *INSERT(NODE("X"),LOC("ABCD",REL("",P=1)))*

The location parameter is identical, but because the checkpoint UUID is empty the new node is simply appended to the target node. In this case the conflict is resolvable and both operations can be executed without losing the intentions of one of the users. The result is that both new nodes are inserted in the total order of their operations.

In the second case the concurrent operations O_a and O_b, for example, are defined as:

O_a = *INSERT(NODE("X"),LOC("ABC",REL("ABCD",P=0)))*

O_b = *INSERT(NODE("Z"),LOC("ABC",REL("ABCD",P=0)))*

Both operations target exactly the same location in the document. With CMAX this conflict is resolvable, because the locations are specified relative to a checkpoint node. Thus the problem can be solved by simply executing the operations in their total order. The result is that both nodes are inserted and the intentions of the users are preserved. The user who generated O_a will notice that some other user has concurrently placed a new node between his node containing the letter *"X"* and the node with the *UUID="ABCD"*. He might consider moving his node to another position if it is necessary but his intention to insert a node "before" the node with the *UUID="ABCD"* is still preserved.

3.2.4.2. Conflicting insert and delete operations

In the case of concurrent insert and delete operations, the resolution is not as simple. Consider two operations O_a and O_b defined as:

O_a = *INSERT(NODE("X"),LOC("ABC",REL("ABCD",P=0)))*

O_b = *DELETE(NODEID("ABC"))*

The operation O_b deletes the node with the *UUID="ABC"* which is the same target node as of O_a. When executing the operations in their total order and $O_a \Rightarrow O_b$, no conflict exists in a syntactical sense. The new node of O_a is inserted at the specified location and after that the node with the *UUID="ABC"* is removed from the document.

It may be argued that the user who issued O_a is puzzled by the sudden removal of the target node. This problem cannot be easily solved and a number of conflict resolution schemes exist for such a case. These are discussed in the section 3.2.5. However, the Collaborative Editing Framework for XML allows the specification of conflict resolution schemes that suit the applied work flow. How this is done is discussed in chapter 5.

Due to the findings on the probability of conflicts in chapter 2, it was decided in favour of the default implementation of CMAX, to let the users solve this problem.

The concurrent operations in the above case are executed in their total order. In the case where $O_b \Rightarrow O_a$, the execution would lead to a problem. When the node with the *UUID="ABC"* is deleted first, the insert operation will fail because its target node is not part of the document any more. In CMAX this is solved by changing the order of the concurrent operations in the history buffer at all sites. This is achieved by modifying the operations state vector so that the total order of the concurrent operations is transformed to $O_a \Rightarrow O_b$. Now the operations can be executed in the same way as above. The modification of the state vector is a new alternative to operational transformation which has been given the title 'state vector swapping' (SVS). The operation itself is not changed, only the state vector of the operation is transformed. As SVS is applied at all sites on the same operations and in the same manner it can be shown that the resulting shared document at all sites are consistent and the syntactical intentions are preserved. The SVS algorithm is discussed in the next chapter.

The same problem as discussed above occurs when a delete operation removes a node from the document that contains the target node of the insert operation as a child in the sub tree. Conflicts such as these are characteristic for hierarchically structured data models such as in XML. In a linear data structure (for example plain

text) these problems do not occur.

Another conflict situation arises in the following case of concurrent insert and delete operations:

O_a = *INSERT(NODE("X"),LOC("ABC",REL("ABCD",P=0)))*

O_b = *DELETE(NODEID("ABCD"))*

In this case the delete operation removes the checkpoint node of the insert operation. If the delete operation is executed before the insert operation after ordering the operations, the insert operation will fail because the checkpoint node is no longer a child node of the node with the *UUID="ABCD"*. This problem is solved in the same way as above by state vector swapping.

3.2.4.3. Conflicting delete operations

From an users perspective, the concurrent delete of the same target node is not a conflict. However, from the technical perspective it is. Consider two concurrent delete operations O_a and O_b:

O_a = *DELETE(NODEID("ABCD"))*

O_b = *DELETE(NODEİD("ABCD"))*

A problem occurs here when executing the operations in their total order. After the first of the two operations is executed the second execution will fail, because the target node has already been removed from the document. The solution here is to discard one of the concurrent delete operations. For example if $O_a \Rightarrow O_b$, O_b is marked as discarded and the execution will have no effect. Discarding an operation does not change the operation itself, or its state vector. The operation O_b is treated like any other operation, the only difference is that the execution will not have an effect.

A similar problem occurs if one of the deleted nodes is part of a sub tree of the other deleted node. In this case the same method is applied as in the case of concurrent insert and delete operations. Consider the operations:

O_a = *DELETE(NODEID("ABCD"))*

O_b = *DELETE(NODEID("ABC"))*

The node with the *UUID="ABCD"* is a child of the node with the *UUID="ABC"*. In the case of $O_a \Rightarrow O_b$ the operations do not conflict and can be executed as is. In the case of $O_b \Rightarrow O_a$ the state vector swapping method is applied before executing the operations.

3.2.5. Conflicting mutational operations

Besides the basic insert and delete operations, CMAX supports update operations which are used for changing the properties of a node. They are called mutational operations, because they mutate a node, in contrast to structural operations which do not change the node but only its location within the document.

The CMAX algorithm is capable of satisfying the properties of convergence, causality preservation and intention preservation on structural operations. However, preserving the intentions of mutational operations can not be attained in all cases. The following example illustrates this:

Consider two concurrent update operations O_a and O_b modifying the same attribute A of the same node N to different values. Let N be a circle element node in an SVG document and A be the radius attribute r with a value of 0.5 (r=0.5). O_a modifies A to the new value 0.9. O_b modifies A to the new value 0.4. Assuming that the radius attribute can only have one valid value at a time, these contrary user intentions lead to a conflict when the corresponding remote operations are received at each site. In order to resolve the conflict, different conflict resolution schemes (CRS) can be applied. Common conflict resolution schemes are:

- **No operation.** Both operations are discarded and the changes have no effect. The result is a radius of r=0.5 at both sites. All user's intentions are lost.
- **Solve by priority.** The operation with the higher priority is executed. Different methods exist to define the priority. For example, each user is given a priority when the collaboration begins or the priority is selected by arbitration as in dARB (Ionescu and Marsic 2000). The result is a convergent document, the intention of one of the users is preserved and the other's are lost.
- **Multi-Versioning.** A new version of the document (or of the node) is created for each conflicting operation (Sun and Chen 2002). This has the drawback that at

the end someone has to decide which version is the correct one. In the case of many versions this can become quite complicated. All user's intentions are preserved until a decision on the correct version is made.

- **Merging.** The result of the two operations are merged to a new value. For example the new value could be the average of both conflicting operations. In this case the new value at both sites would be r=0.65. All user's intentions are integrated but depending on the application, this may or may not make sense. In the worst case the result is semantically incorrect and all user's intentions are lost.
- **Solve by semantic context.** One of the values is chosen depending on the semantic context of the application. This could be, for example, the maximum or the minimum value or a completely new calculated value. Important is that the new attribute value makes most sense semantically. Molli, Skaf-Molli et al. (2002) transform the conflicting operations so that the maximum value is always chosen. This could make sense in some applications but is never a general solution to the problem.
- **User Notification.** The users are notified of the conflict and have to decide which value is the correct one. This way of manual conflict solution can be combined with other conflict resolution schemes. All user's intentions are preserved until a decision is made.

CMAX resolves the problem in the following way. Based on the result of the conflict probability studies in chapter 2, it was decided to let the users solve this problem. With CMAX the concurrent update operations are executed in their total order. Thus the resulting value of the radius attribute for the case of

$O_a \Rightarrow O_b$ would be r=0.4. For the case of $O_b \Rightarrow O_a$ the result would be r=0.9.

One of the users intentions is lost and he or she might be surprised, that the value is different than the one he or she changed it to. The user would probably try to change it again. In such a case it is very important that the collaborating users are aware of what the other users are currently working on. This can be achieved by supporting awareness mechanisms which is done so by CEFX and is discussed in chapter 5.

If a user modifies a node and concurrently another user deletes the same node or one of its parents in the document tree, the following happens. CMAX handles this problem the same way as it does with concurrent insert and delete operations.
If $O_b \Rightarrow O_a$ where O_a is the delete operation and O_b is the update operation, the operations are executed in their total order. In the case of $O_a \Rightarrow O_b$, the total order of the conflicting operations is reversed to $O_b \Rightarrow O_a$ and they are executed.

3.2.6. Locking of nodes for conflict prevention

Besides the methods of conflict resolution provided by CMAX, CEFX additionally provides an optional locking scheme. Locking is a simple method to prevent conflicts such as those described above. The CEFX locking scheme is similar to the one described by Galli (2000). In contrast to Galli the locking is optional and the locking granularity is more flexible. The user has the opportunity to select a whole set of nodes and sub-trees and mark them as locked, instead of only locking one selected node and its child nodes (sub-tree). The ability to lock a user-defined part of a document is a desired feature in a collaborative editing application. For example in the case where a various number of users work on one large document, there are times when users want to collaborate freely and other times when they wish to work on their part of the document alone, without being disturbed by another user. This flexibility in the locking granularity is a new feature, which allows locking of parts of a document by one user, while other parts are open for collaborative real-time editing. The locking mechanism is not part of the CMAX consistency maintenance algorithm. Locking is transparently applied by CEFX and independent of the used consistency maintenance algorithm. As CEFX allows integration of different consistency maintenance algorithms, this has the advantage that these algorithms do not require additional effort in supporting lock and unlock operations. How optional locking is achieved in CEFX is discussed in chapter 5.

3.2.7. Informal specification of the CMAX algorithm

The CMAX algorithm was designed for the XML data model as specified by the World Wide Web Consortium (W3C). Recommendation for the Extensible Markup Language (XML) 1.0 (Fourth Edition). For the manipulation of XML this works im-

plementation of CMAX makes use of the Document Object Model (DOM) Level 3 interface.

In the following we present the CMAX algorithm for maintaining consistency over XML documents. This algorithm was applied in the default implementation of CE-FX.

The following notations will be used in the description of the CMAX algorithm:

- O_r represents a new remote operation arriving at a site
- O_l represents a new local operation issued at a site
- HB_s represents the history buffer containing executed operations at a site
- O_i represents the i^{th} executed operation at a site stored in the HB_s
- *COV* represents the concurrent operations vector containing all operations that are identified as concurrent
- *CRHM* represents a map of conflict resolution hints for conflicting operations
- SV_s represents a sites state vector
- SV_o represents an operations state vector
- OQ_s represents the FIFO queue of incoming remote operations at a site
- $O_l \oslash O_r$ means that O_l is in conflict with O_r

Algorithm 1.1. *CMAX_EXECUTE_LOCAL_OPERATION (O_l)*

1. Execute O_l
2. Update SV_s
3. Assign value of SV_s to SV_o
4. Add O_l to HB_s
5. Propagate O_l to all other sites

After executing a local operation, the local state vector is updated by incrementing the operations count for the corresponding site. Then the new value of the local state vector is assigned to the state vector of the operation and the local operation is added to the history buffer, before it is propagated to the other sites.

Algorithm 1.2. *EXECUTE_AND_UPDATE(O)*

1. Execute O
2. Update SV_s
3. Add O to HB_s

When executing an operation, the local state vector is updated and the operation is added to the history buffer.

Algorithm 1.3. *UNDO(O)*

1. Undo O
2. Decrement SV_s
3. Remove O from HB_s

When undoing an operation, the state vector is decremented and the operation is removed from the HB_s.

Algorithm 1.4. *CMAX_EXECUTE_REMOTE_OPERATION (O_r)*

1. Check if O_r is causally ready to execute
2. If O_r is causally ready
    1. If HB_s=*[]*
        1. *EXECUTE_AND_UPDATE(O_r)*
    2. Else if $HB_s=[O_1,...,O_{n-1}]$, where n represents the number of all executed operations at the site
        1. Search for the position p of the last operation $O_i \Rightarrow O_r$ in HB_s
        2. For all O_i in HB_s from $i=p$ to $p=n-1$ add O_i to *COV*
        3. Add O_r to *COV*
        4. Order all operations in *COV* after total order
        5. For O_i in HB_s from n-1 to p *UNDO(O_i)*
        6. Check all O_i in *COV* for a conflict with O_r
            1. If $O_i \oslash O_r$ find a conflict resolution hint (*CRH)* for the conflict and add *CRH* to *CRHM*
        7. For all $O_i \oslash O_r$ in *COV* apply the corresponding *CRH* from *CRHM* to resolve the conflict

8. Order all operations in *COV* after total order
9. For all operations in *COV EXECUTE_AND_UPDATE(O)*

3. Else, add O_r to OQ_s

First the remote operation is checked if it is causally ready for execution. This is done by comparing the state vector of the operation to the local state vector as described in Definition 1: conditions for executing remote operations. If the operation is not causally ready, this means that another operation that is causally preceding has not been received yet and the operations execution has to be delayed until the missing operation is executed. Incoming remote operations that are not causally ready are therefore stored in a queue (OQ_s). The operations in the queue are then processed later. When a causally ready operation is the first operation to be executed at a site, the history buffer is still empty and the operation can be executed directly as described in Algorithm 1.2. When the history buffer HB_s contains operations it has to be checked to establish if the new operation (O_r) is conflicting with any of the concurrent operations in HB_s. This is done by applying the following procedure:

All operations that are concurrent (independent) with the new operation are selected and stored in the concurrent operations vector (*COV*). The new operation is added to the *COV* and then the operations in the *COV* are ordered, so that the new operation is in the correct total order position within the other operations in the *COV*. After that all operations in HB_s which are concurrent with O_r are undone (see *undo/do/redo* scheme). The next step is to check for conflicts of O_r with concurrent operations.
If a conflict is found, a conflict resolution hint (*CRH*) is selected for the type of conflict and stored in the map for conflict resolution hints (*CRHM*). The *CRHM* maps an operation in *COV* to its corresponding *CRH*. The *CRH* is a data structure that identifies a certain action that is to be undertaken in order to solve a certain conflict. The types of conflicting operations and their conflict resolutions were discussed in the sections 3.2.4 and 3.2.5. The *CRHs* used in this works implementation are discussed further in the next chapter. After applying the *CRHs* for all conflicting operations in *COV*, the operations in *COV* are ordered again, so that state vector swapping changes have an effect on the total order of operations. In a last step all operations in *COV* are executed, the local state vector is updated and the operations are added to the HB_s.

By executing the operations in the *COV*, the new operation is executed and all undone operations are redone.
The next chapter discusses the software implementation of operations in CMAX.

3.2.8. Conclusions

The work described in this thesis is the first documented attempt of the successful use of universally unique identifiers (UUIDs) in a consistency maintenance algorithm to address nodes within an XML document and to use a new state vector swapping scheme (SVS) in order to preserve intentions. Using UUIDs has advantages over positional addressing schemes. Intentions can be preserved without the need to transform operations, reducing complexity and calculation time.
In the case of resolvable conflicts the SVS scheme was successfully used to solve the conflicts instead of applying operational transformation (OT), which is already used in other contemporary consistency maintenance algorithms. The consistency maintenance algorithm (CMAX) supports the synchronisation of any XML document type. As XML document types exist for many application areas, the CMAX algorithm can be generally used for the synchronisation of, for example, 2D and 3D graphic documents as well as text documents or spreadsheets.

Chapter 4. Implementation of the algorithm in software

For the implementation of the CMAX algorithm in software, the Java programming language was chosen. Java is a widely used, platform independent[36] and object-oriented language. The standard Java Runtime provides libraries for XML parsing and processing[37], reducing implementation effort. These were the main reasons for choosing Java for the implementation of CMAX. This chapter describes the main components of the software, their roles and tasks within CEFX and how the testing of the software components was conducted. The source code that is discussed in this chapter and attached to this thesis, represents the formal specification of the CMAX algorithm.

4.1. Software components

The implementation of CMAX is split up into three components. The main part of the algorithm is placed within the concurrency controller (CC) component. The concurrency controller inquires the conflict resolution provider (CRP) in the case of a conflict for a conflict resolution hint (CRH) as described in chapter 3. The manipulation of an XML document is conducted by the `Operation` components. `Operation` objects are transmitted between the collaborating sites and hold all necessary information on how to execute an operation that has been issued by a user. The following sections discuss these three components.

4.1.1. Concurrency Controller

The concurrency controller implementation is split up into a number of classes and interfaces. The `ConcurrencyController` interface defines the basic methods of the concurrency controller. An abstract base class, the `AbstractConcurrencyControllerImpl` (ACCI) class, implements most of the interface methods and leaves only a few methods to be implemented for the subclasses (in this case the `OrderingConcurrencyControllerImpl` class). The ACCI class has been developed in order to

---

[36] Java Virtual Machine implementations exist for many operating systems such as Windows, Mac OS, Linux and Unix

[37] The Java 2 Standard Edition (J2SE) contains the Apache Xerces XML parser

simplify the development of third party concurrency controller components. The `OrderingConcurrencyControllerImpl` is the default CC implementation used in CEFX. The `AbstractConcurrencyControllerImpl` class also implements the methods defined by the `ExecutionContext` interface. This interface is necessary when executing an operation, which is explained later.

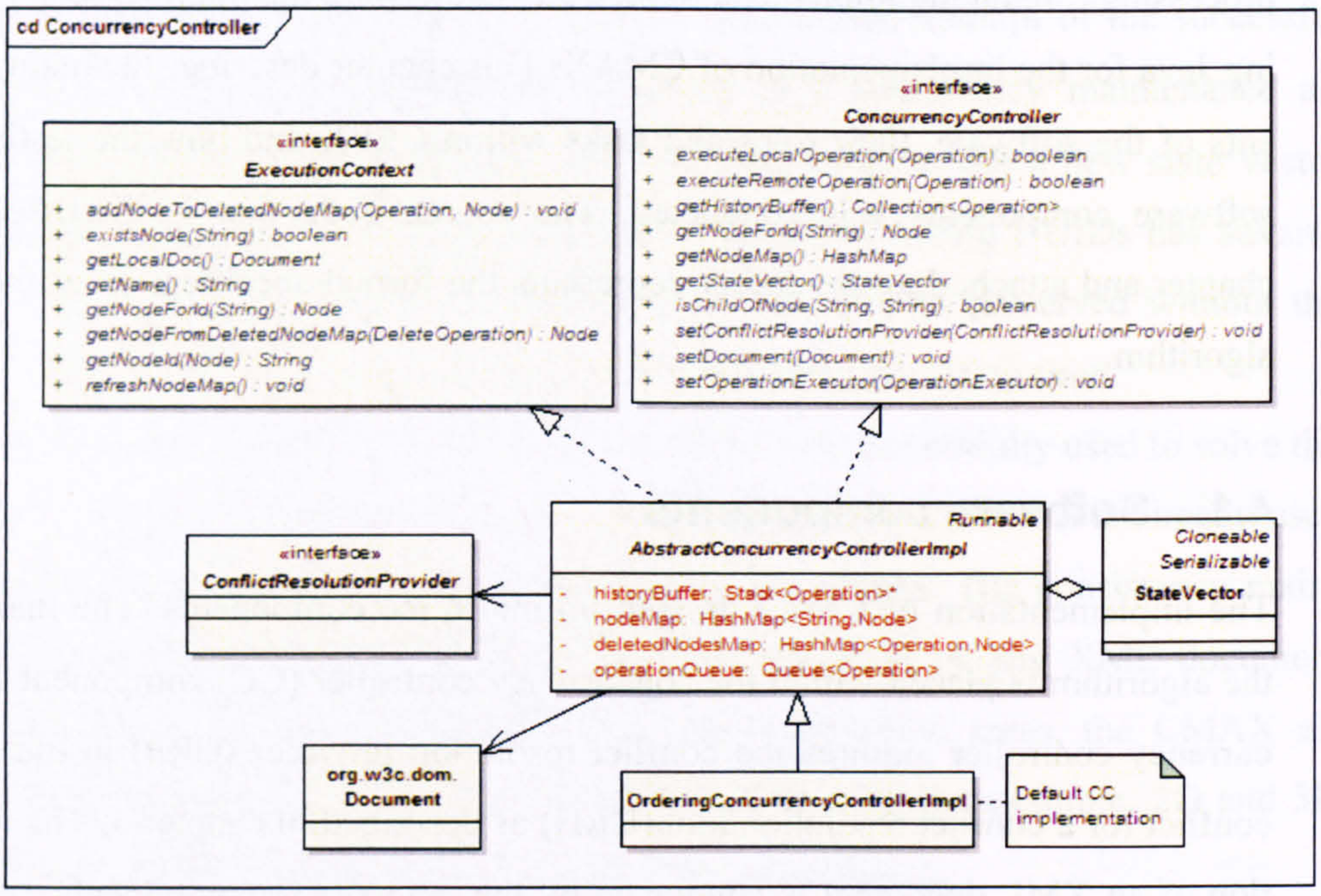

Figure 4.1: Concurrency Controller class diagram

As shown in Figure 4.1, the ACCI class has a history buffer (`historyBuffer` property of type `Stack`) containing operations (`Operation` objects). It has an operation queue (property `operationQueue`), representing a queue of operations that are ready to be executed. The ACCI holds a map of nodes (`nodeMap` property of type `HashMap`) mapping all nodes of the current document to their UUID (type `String`). The node map facilitates to find a node within the document very quickly on basis of its UUID. The ACCI additionally has a map of deleted nodes (`deletedNodeMap` property) that is used to store all nodes that were deleted from a document. The ACCI aggregates a state vector (`StateVector`). The `StateVector` class implements the interfaces `Clonable` and `Serializable` which is required in order to be able to send a copy of a `StateVector` over the network. The ACCI references objects of the type `ConflictResolutionProvider` and `org.w3c.dom.Document`. The

`org.w3c.dom.Document` interface is the DOM interface for the manipulation of the current XML document.

4.1.1.1. ConcurrencyController interface methods

The `ConcurrencyController` interface declares the following methods:

- `boolean executeLocalOperation(Operation operation);`
- `boolean executeRemoteOperation(Operation operation);`
- `StateVector getStateVector();`
- `Collection<Operation> getHistoryBuffer();`
- `void setDocument(Document localDoc);`
- `void setConflictResolutionProvider(ConflictResolutionProvider crp);`
- `void setOperationExecutor(OperationExecutor impl);`

The methods `executeLocalOperation(Operation operation)` and `executeRemoteOperation(Operation operation)` are called by the `CEFXController` if either the user issued a local operation, or a new remote operation was received over the network from another site. The method `getStateVector()` is used for retrieving the current local state vector and setting the value of the state vector of a new operation to the current state vector value. The method `getHistoryBuffer()` is used for clearing the history buffer after loading a document and initialising a new editing session. The method `setDocument(Document localDoc)` provides the concurrency controller with a reference to the new document which is to be taken under concurrency control. After setting the document reference, the concurrency controller analyses the document and starts to index every node by adding each node in the document to the controller's map of nodes (`nodeMap`). The method

`setConflictResolutionProvider(ConflictResolutionProvider crp)` is called by the `CEFXController` in order to provide the `ConcurrencyController` with a reference to the `ConflictResolutionProvider`.

The method `setOperationExecuter(OperationExecutor impl)` provides the

ConcurrencyController with a reference to the OperationExecutor object. This method is called by the CEFXController when initialising the ConcurrencyController. The OperationExecutor is an interface to the client, in this case represented by the CEFXController, providing information on the client's name and identifier. It also allows notification of the client if an issued operation is not supported. It depends on the implementation of the ConcurrecyController if an operation is supported or not. In this case the ConcurrencyController, for example, does not support an operation that would delete the root node of a document. Deleting the root node of a document would lead to an invalid document and all subsequent operations would become invalid. The OperationExecutor interface also provides a method that allows notification of the client on real conflicts. A real conflict is one that can not be solved automatically by the ConcurrencyController implementation. The client (CEFXController) can then use this information to notify the user of the situation and, for example, let the user solve the problem.

4.1.1.2. ExecutionContext interface methods

The context of execution is the local document state at each editing site. The ExecutionContext interface declares methods that provide information on the current context of execution and allows modification of it. Operations that are executed use those methods to achieve the required effect and thus change the context. The methods declared by ExecutionContext are as follows:

- `Node getNodeForId(String uuid);`
- `String getNodeId(Node node);`
- `boolean isChildOfNode(String childNode, String parentNode);`
- `boolean existsNode(String nodeUuid);`
- `Document getLocalDoc();`
- `void refreshNodeMap();`
- `Node getNodeFromDeletedNodeMap(Operation op);`
- `Node addNodeToDeletedNodeMap(Operation op, Node node);`

The method `getNodeForId(String uuid)` retrieves the node with the given UUID from the document. When an operation is transmitted over the network the target

node of an operation is identified by the UUID. This method is used to get a reference to the node object within the local document. The method `getNodeId(Node node)` retrieves the UUID of a given node in order to, for example, compare it with an UUID of another node. The method `isChildOfNode(String childNode, String parentNode)` is called by the `ConflictResolutionProvider`. It is used to find out, if a node with the given UUID in the `childNode` argument is part of a sub tree starting from the node with the UUID in the `parentNode` argument. This is necessary in order to know if a deletion of a node might interfere with an insertion or an update on another node which is part of the sub tree.

In order to find out if a node with a certain UUID still exists in the local context, the method `existsNode(String nodeUuid)` is used. The method `getLocalDoc()` provides the caller with a reference to the current local document object. Operations use this method to modify the document directly.

The `ExecutionContext` (represented by the concurrency controller implementation) owns a map of all nodes in the document (`nodeMap`), as mentioned before. After an operation is executed, the operation calls the method `refreshNodeMap()` so that all changes to the document are reflected in the map of nodes. That is for example, if a new node has been added to the document, the node map is updated and the new node is added to the `nodeMap`. A second map of nodes is used to store all nodes that were deleted from the document (`deletedNodesMap`). This is done in order to be able to undo a delete operation easily.

The method `getNodeFromDeletedNodeMap(Operation op)` is used to retrieve a node from the map of deleted nodes. The method `addNodeToDeletedNodeMap(Operation op, Node node)` is used to store a deleted node in the `deletedNodesMap` of the concurrency controller implementation.

4.1.1.3. The AbstractConcurrencyControllerImpl class

The concurrency controller used in this work was implemented in two steps. In the first step the abstract base class `AbstractConcurrencyControllerImpl` (as shown in figure 4.1) was implemented. The second step was to implement the `OrderingConcurrencyControllerImpl` class. This section explains the functionality that is

implemented in the `AbstractConcurrencyControllerImpl` class.

The concurrency controller implementation is instantiated by the `CEFXController`. How this is done is explained in chapter 6. After instantiating the concurrency controller, it is initialised. The abstract class `AbstractConcurrencyControllerImpl` therefore provides the `init(OperationExecutor client)` method. When this method is called, the following steps are executed:

- Store the reference to the client (the `CEFXController`) in a field of the type `OperationExecutor`
- Create a new empty history buffer
- Create a new empty map of nodes (`nodeMap`)
- Create a new empty map of deleted nodes (`deletedNodesMap`)
- Create a new empty state vector
- Create a new empty operation queue
- Start the concurrency controller thread

The `AbstractConcurrencyControllerImp` class is implemented as a thread running in parallel to the editing application. The thread runs in an endless loop and checks after a certain delay time (in this case every 100 milliseconds) if an operation in the operation queue exists, that is ready for execution. The activity diagram in figure 4.2 shows the flow of activities in the `run()` method of the `AbstractConcurrencyControllerImpl` class.

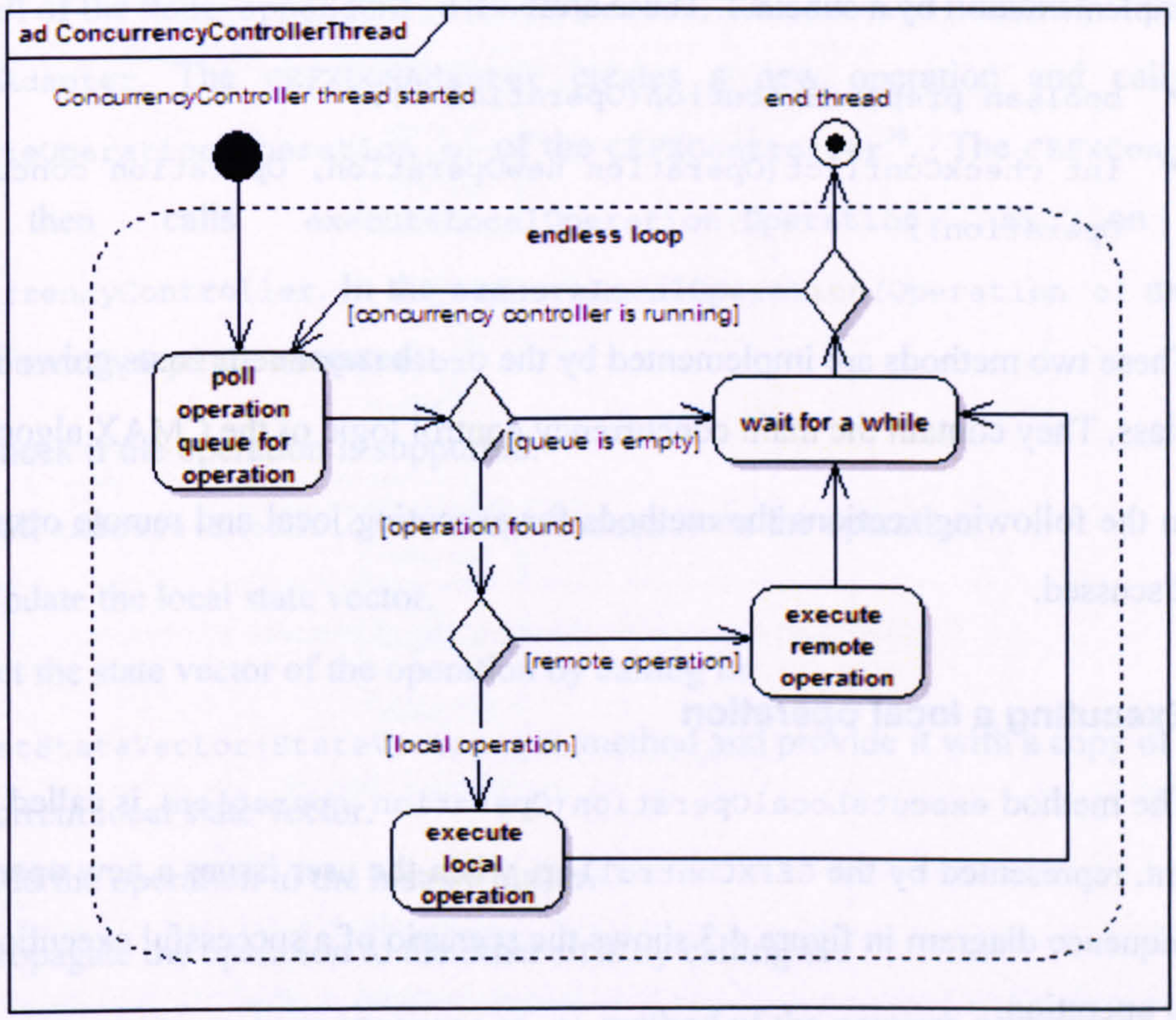

Figure 4.2: ConcurrencyController thread activities

Each operation found in the operations queue is checked if it is a local or a remote operation. This is done by comparing the operation's and the client's (the `OperationExecutor`'s) id. If the ids are identical the operation is local, otherwise it is a remote operation. Remote operations are executed by calling the `executeRemoteOperation(Operation op)` method. In the case of a local operation, it is executed by calling the `executeLocalOperation(Operation op)` method.

The `AbstractConcurrencyControllerImpl` class implements all methods of the `ConcurrencyController` interface except the method `setOperationExecutor(OperationExecutor impl)`. This method is implemented by its subclass, the `OrderingConcurrencyControllerImpl` class. All methods of the `ExecutionContext` interface are implemented completely by the `AbstractConcurrencyControllerImpl` class.

`AbstractConcurrencyControllerImpl` declares two abstract methods that require

implementation by a subclass. These are:

- `boolean prepareExecution(Operation o);`
- `int checkConflict(Operation newOperation, Operation concurrent-Operation);`

These two methods are implemented by the `OrderingConcurrencyControllerImpl` class. They contain the main concurrency control logic of the CMAX algorithm.

In the following sections the methods for executing local and remote operations are discussed.

Executing a local operation

The method `executeLocalOperation(Operation operation)` is called by the client, represented by the `CEFXController`, when the user issues a new operation. The sequence diagram in figure 4.3 shows the scenario of a successful execution of a local operation.

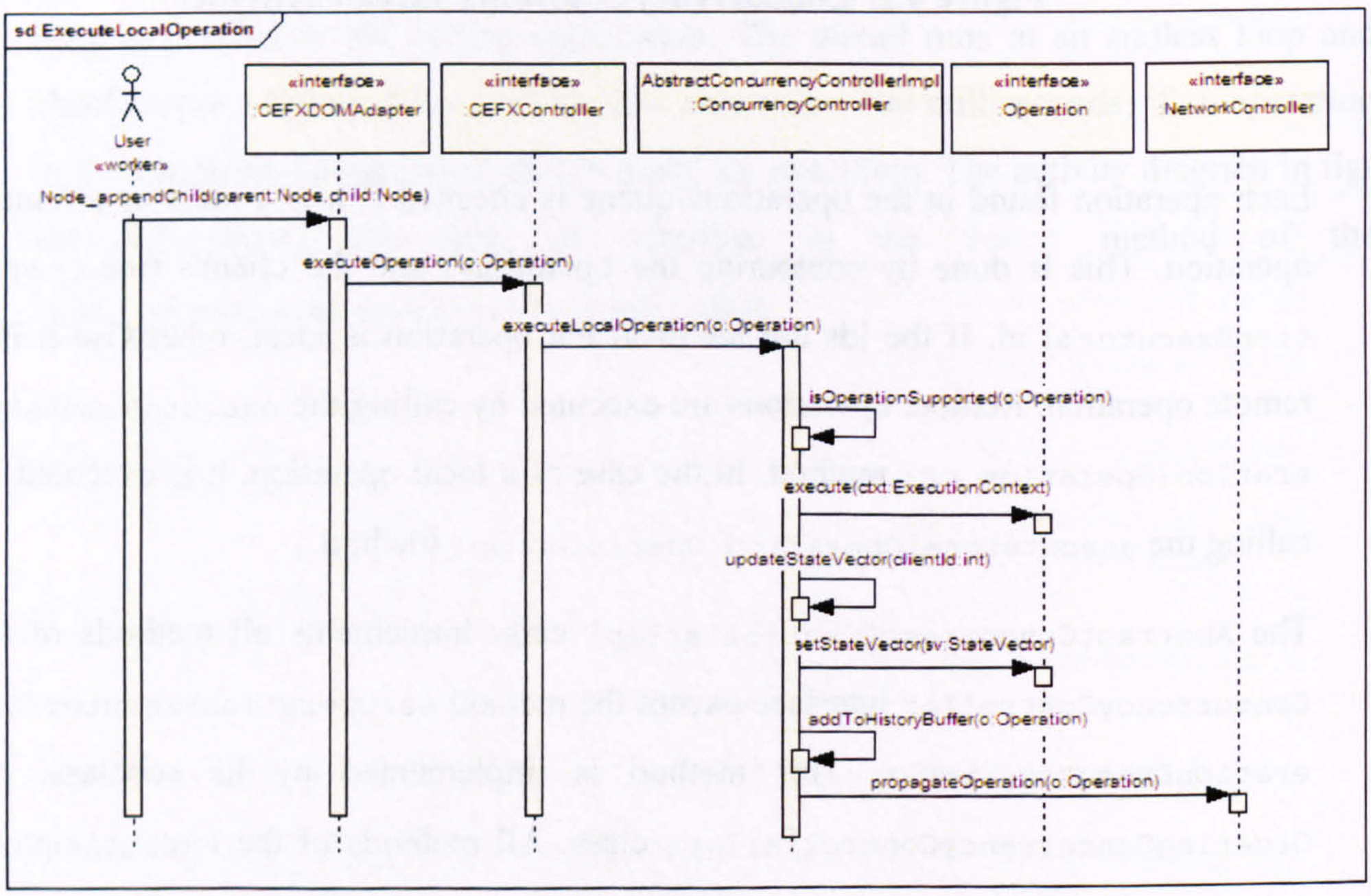

Figure 4.3: Execution of a local operation scenario

In this scenario the user creates, for example, a new node in the document. This leads

to a call of the `Node_appendChild(Node parent, Node child)` method of the `CEFXDOMAdapter`. The `CEFXDOMAdapter` creates a new operation and calls the `executeOperation(Operation o)` of the `CEFXController`[38]. The `CEFXController` then calls `executeLocalOperation(Operation o)` on the `ConcurrencyController`. In the `executeLocalOperation(Operation o)` method the following steps are executed:

- Check if the operation is supported.
- Call `execute(ExecutionContext context)` on the operation.
- Update the local state vector.
- Set the state vector of the operation by calling its `setStateVector(StateVector sv)` method and provide it with a copy of the current local state vector.
- Add the operation to the history buffer.
- Propagate the operation to the other sites by calling the `propagateOperation(Operation o)` method of the network controller[39].

Executing a remote operation

Figure 4.4 shows a scenario with the successful execution of a remote operation. Remote operations are received by the network controller of CEFX. After the network controller received an operation from another site, it calls the `executeRemoteOperation(Operation o)` of the `CEFXController`. The `CEFXController` then calls the `executeRemoteOperation(Operation o)` of the `ConcurrencyController`.

---

[38] The CEFXDOMAdapter and the CEFXController are discussed in chapters 5 and 6.

[39] The NetworkController is discussed in chapter 5 and 6

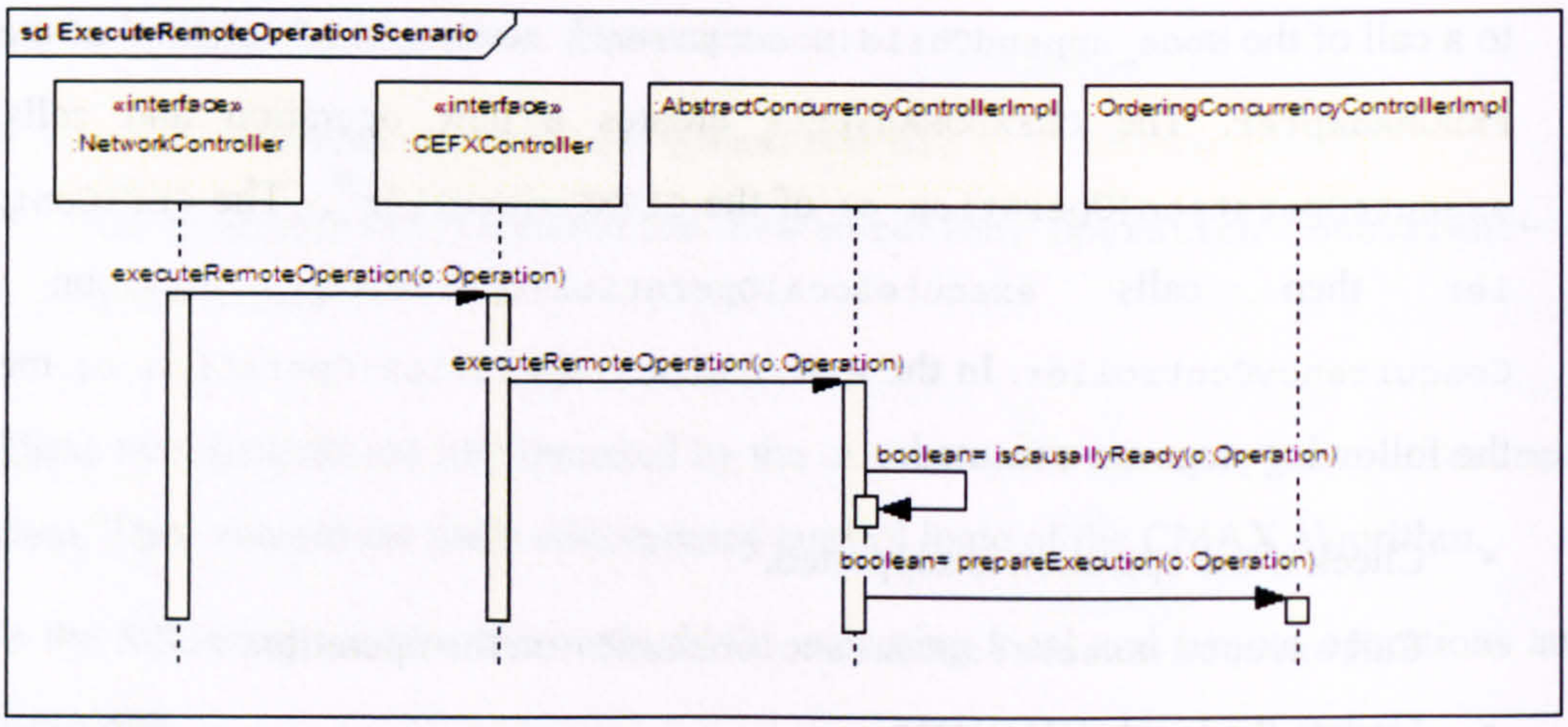

Figure 4.4: Scenario of executing a remote operation

The `ConcurrencyController` is, as mentioned before, implemented by the `AbstractConcurrecyControllerImpl` and the `OrderingConcurrencyControllerImpl` classes. In the sequence diagram in figure 4.4 both classes are depicted to show which method is implemented by which class. In reality only an object of the `OrderingConcurrencyControllerImpl` type is instantiated. The `ConcurrencyController` checks if the operation is causally ready for execution. If the operation is causally ready the method `prepareExecution(Operation o)` of `ConcurrencyController` is called.

The execution steps within the `executeRemoteOperation(Operation o)` method of the `AbstractConcurrencyControllerImpl` are shown in the activity diagram in figure 4.5. First the operation is checked if it is causally ready by calling the method `isCausallyReady(Operation o)` (also see figure 4.4). If it is not causally ready, the operation is appended to the operation queue for later execution. If it is causally ready, the `prepareExecution(Operation o)` method, which is implemented by the `OrderingConcurrencyControllerImpl` class, is called. If the operations execution failed, the operation is appended to the operation queue and handled at a later time in the method `run()` of the `AbstractConcurrencyControllerImpl` thread.

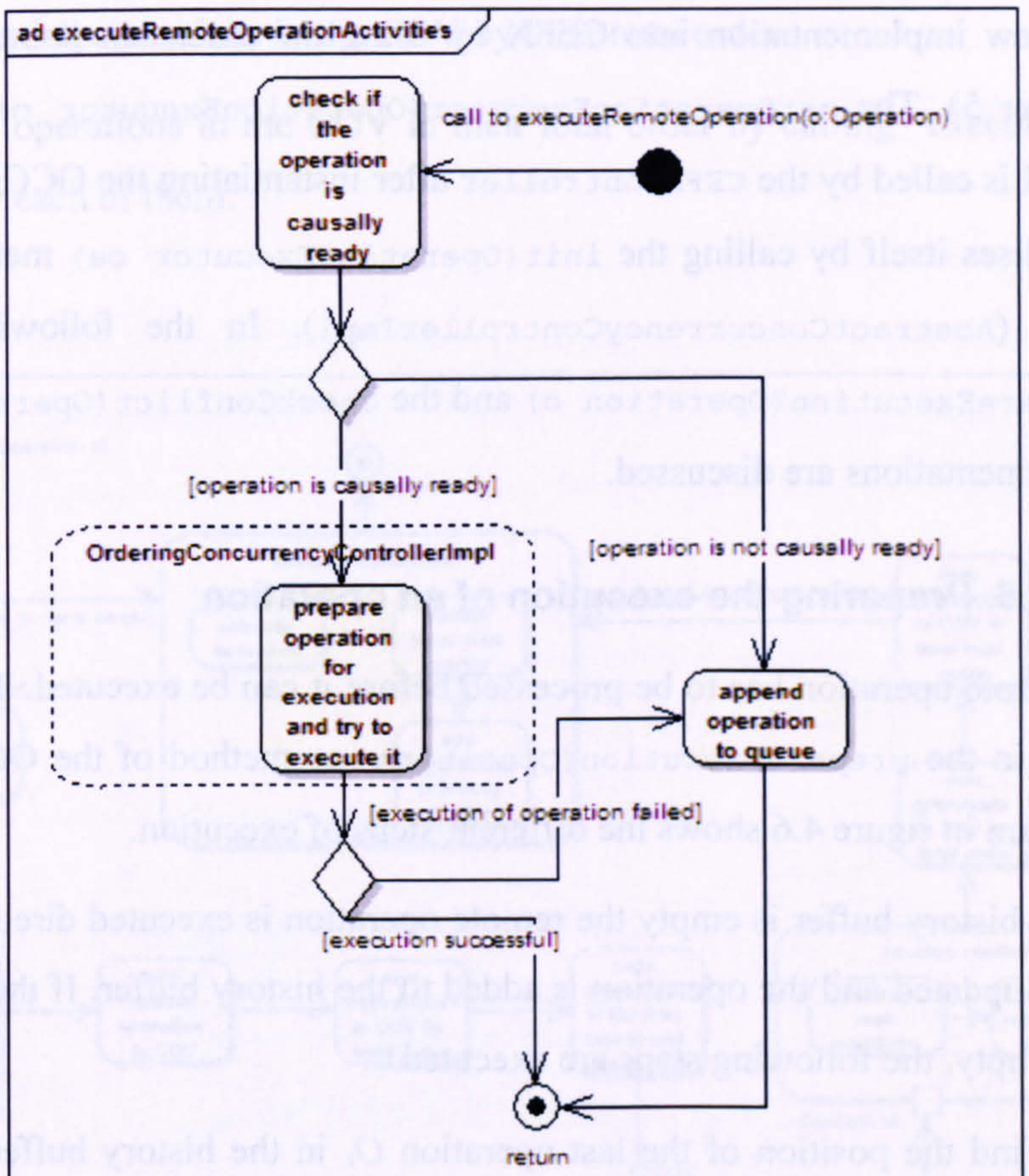

Figure 4.5: Executing a remote operation activity diagram

The activities shown in figure 4.5 correspond to the steps 1. and 3. of the CMAX algorithm specification (see algorithm 1.4. in chapter 3.2.7). The "prepare operation for execution and try to execute it" activity - which is implemented by the `OrderingConcurrencyControllerImpl` class - corresponds to step 2. of algorithm 1.4.

4.1.1.4. The OrderingConcurrencyControllerImpl class

The methods `prepareExecution(Operation o)` and `checkConflict(Operation o)` are implemented by the `OrderingConcurrencyControllerImpl` (OCCI) class. Additionally the OCCI implements the `setOperationExecutor(OperationExecutor oe)` method of the `ConcurrencyController` interface. Any concurrency control implementer would only need to implement those three methods and hook

the new implementation into CEFX by using the extension point mechanism (see chapter 5). The `setOperationExecutor(OperationExecutor oe)` method of the OCCI is called by the `CEFXController` after instantiating the OCCI. The OCCI then initialises itself by calling the `init(OperationExecutor oe)` method of its super-class (`AbstractConcurrencyControllerImpl`). In the following sections the `prepareExecution(Operation o)` and the `checkConflict(Operation o)` method implementations are discussed.

4.1.1.5. Preparing the execution of an operation

A remote operation has to be processed before it can be executed. This processing is done in the `prepareExecution(Operation o)` method of the OCCI. The activity diagram in figure 4.6 shows the different steps of execution.

If the history buffer is empty the remote operation is executed directly, the state vector is updated and the operation is added to the history buffer. If the history buffer is not empty, the following steps are executed.

1. Find the position of the last operation O_l in the history buffer that totally precedes the remote operation O_r ($O_l \Rightarrow O_r$).
2. Copy the tail of the history buffer into a list of concurrent operations (concurrent operations vector, COV), starting from the position of the last totally preceding operation up to the last operation in the history buffer.
3. Add the remote operation to the COV.
4. Sort all operations in the COV by their total order.
5. Undo all operations in the history buffer starting at the end of the history buffer up to the last totally preceding operation.
6. Find all conflicts of the operations in COV (input argument of "find and classify conflicts") with the remote operation and classify them. The result is a list of conflicts (`ConflictList`).
7. Resolve all conflicts in the list of conflicts. First process the real conflicts (if any) then process the resolvable conflicts.

8. Again sort all operations in the COV by their total order.
9. Redo all operations in the COV in their total order by calling "execute and update" for each of them.

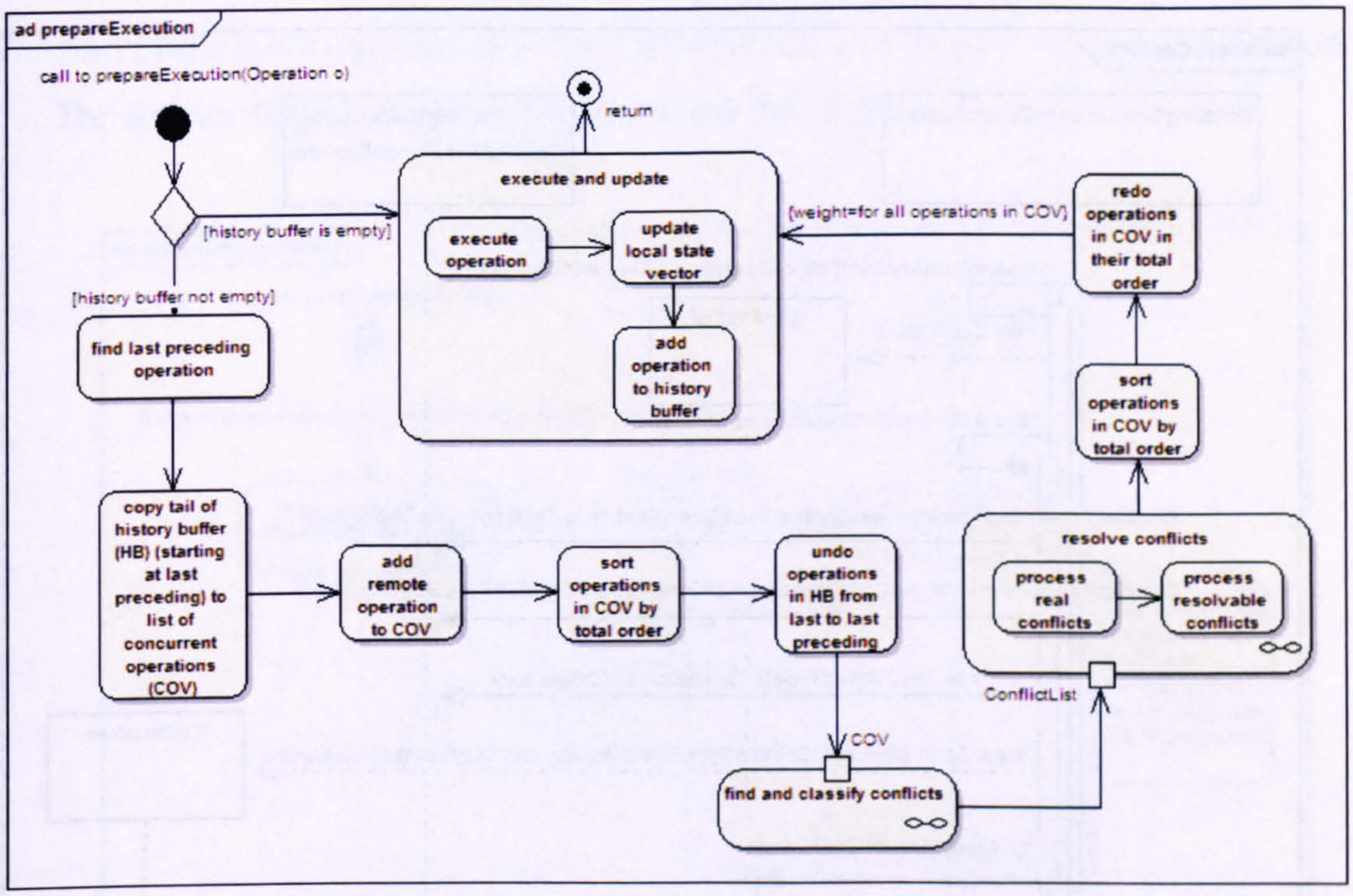

Figure 4.6: Prepare execution activity diagram

Finding and classifying conflicts

Each remote operation is checked for a possible conflict. This is done in the method `findAndClassifyConflicts(Operation o, List<Operation> concurrentOperations)` in the OCCI. For each operation in the COV the `checkConflict(Operation o, Operation co)` method is called. The first argument of this method is a reference to the remote operation. The second argument is a reference to the concurrent operation from the COV. In this method the `ConflictResolutionProvider` (CRP) is called in order to retrieve a `ConflictTypeSpecification` (CTS), a `ConflictResolutionHint` (CRH) and a conflict type. The `ConflictTypeSpecification` indicates the reason for the conflict

and the `ConflictResolutionHint` indicates how the conflict should be resolved. With the help of this information the operations are classified into resolvable conflicting, unresolvable conflicting and not conflicting operations. This classification is indicated by the conflict type.

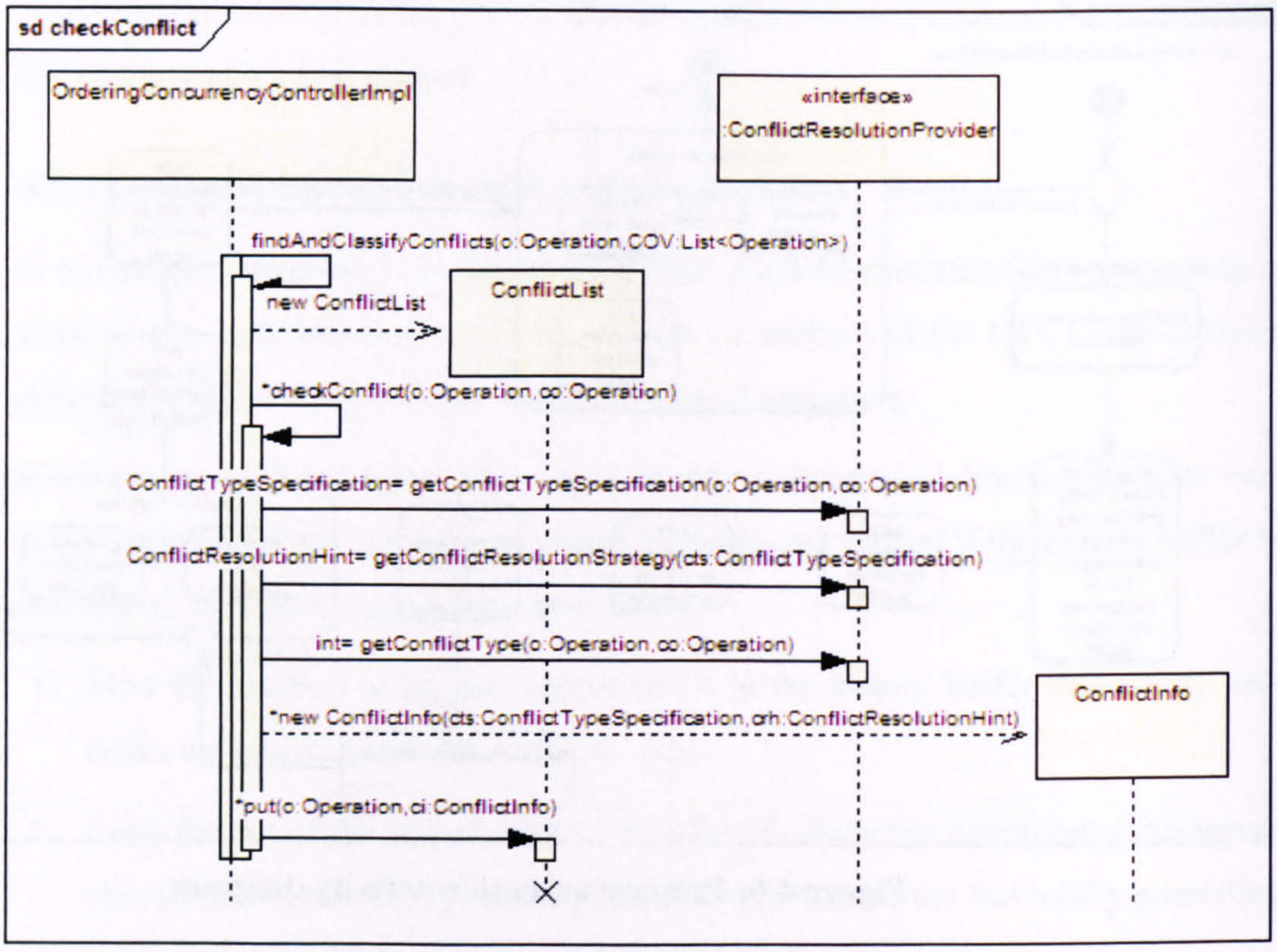

Figure 4.7: Check conflict sequence diagram

For each conflict type a map is created that maps each operation to its specific CTS and CRH[40]. The CTS and the CRH are thereby stored in a `ConflictInfo` object. The result of this classification is a `ConflictList` containing the maps of operations for each conflict type. The sequence diagram in figure 4.7 shows the calls to the `ConflictResolutionProvider`.

Resolving conflicts

After all conflicts have been found and classified, they need to be resolved. In the best case no unresolvable or "real" conflicts exist. In the default implementation of CEFX all possible conflicts are resolved and thus no "real" conflicts can occur.

[40] CRP, CTS and CRH are further discussed in chapter 4.1.2

However, the `ConflictResolutionProvider` was designed so that it can be replaced by a different implementation that knows cases of unresolvable conflicts. This may make sense in other work flows where, for example, the users should be notified directly if a certain conflict occurs.

In the implementation of OCCI, the "real" conflicts are handled in the `processRealConflicts(List<Operation> cov, Operation ro, ConflictList cl)` method. The activity diagram in figure 4.8 shows how this is done.

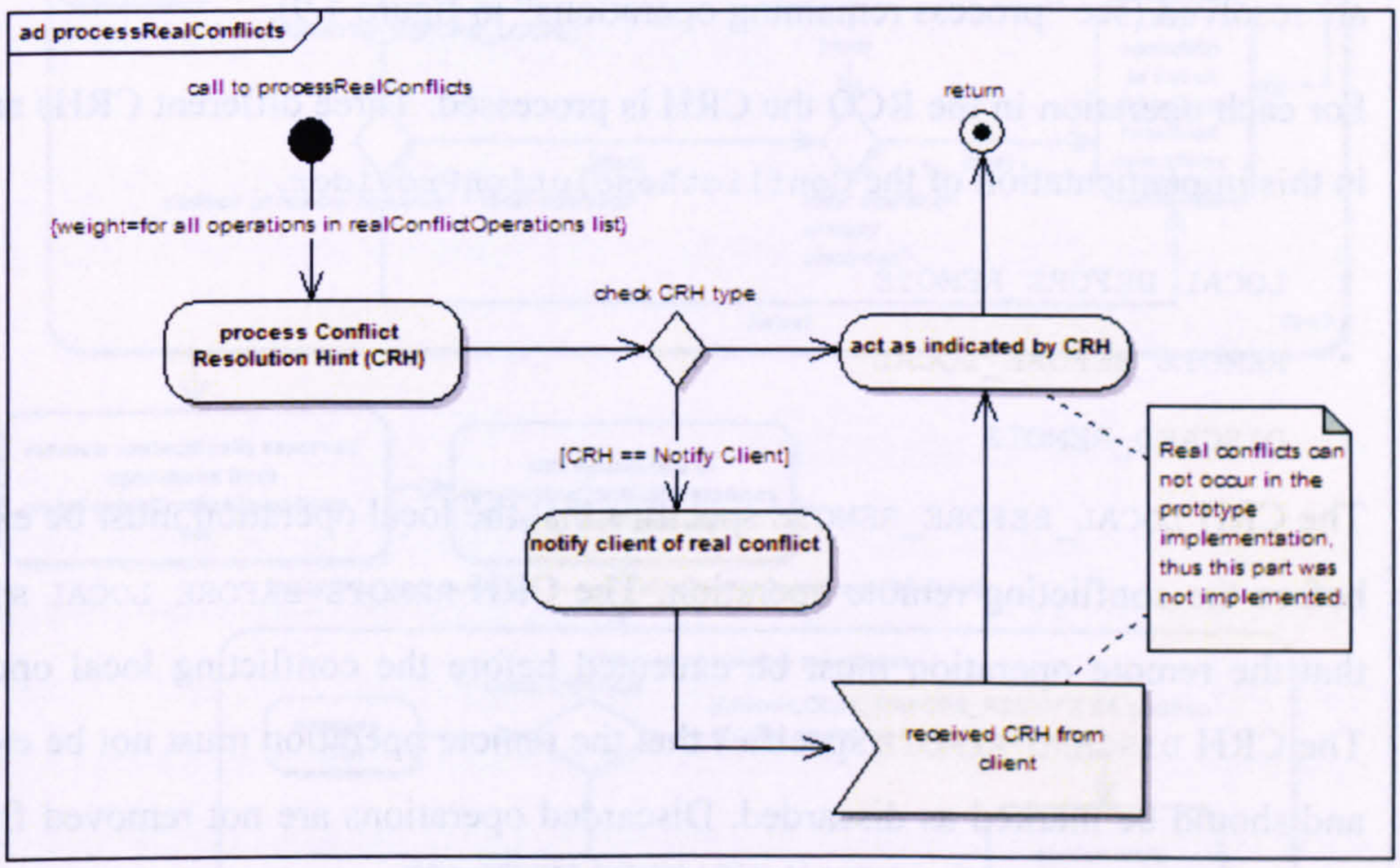

Figure 4.8: Process real conflicts activity diagram

If a "real" conflict exists the CRH indicates to notify the client. This is done by calling the `notifyOfRealConflict(ConflictTypeSpecification cts, Operation o, Operation co)` method of the `OperationExecutor` which returns a `ConflictResolutionHint`. After receiving the CRH from the client the corresponding action, as indicated by the CRH, should be taken. However, as this case cannot occur in the default implementation of CEFX, this was left open to third party developers using a different `ConflictResolutionProvider` implementation.

Resolvable conflicts are processed in the `processResolvableConflicts(List<Operation> cov, Operation o, ConflictList cl)` method of the OCCI. Figure 4.9 shows how these conflicts are processed. When the `processResolvableCon-`

flicts(...) method is called, the list of concurrent operations (COV), the remote operation and the list of conflicting operations are passed as arguments. The ConflictList contains a map of resolvable conflict operations (RCO) and their corresponding CRH. All operations in the RCO are then processed in two steps. First all operations are identified that are in conflict with the remote operation, but the conflict is resolved automatically (see "identify operations with automatically resolved conflicts" in figure 4.9). These operations are then stored in a list of automatically resolved operations (ARO). In the second step the remaining conflicts are resolved (see "process remaining operations" in figure 4.9).

For each operation in the RCO the CRH is processed. Three different CRHs are used in this implementation of the ConflictResolutionProvider.

- LOCAL_BEFORE_REMOTE
- REMOTE_BEFORE_LOCAL
- DISCARD_REMOTE

The CRH LOCAL_BEFORE_REMOTE specifies that the local operation must be executed before the conflicting remote operation. The CRH REMOTE_BEFORE_LOCAL specifies that the remote operation must be executed before the conflicting local operation. The CRH DISCARD_REMOTE specifies that the remote operation must not be executed and should be marked as discarded. Discarded operations are not removed from the history buffer and are treated as normal operations. The only difference to normal operations is that their execution will have no effect.

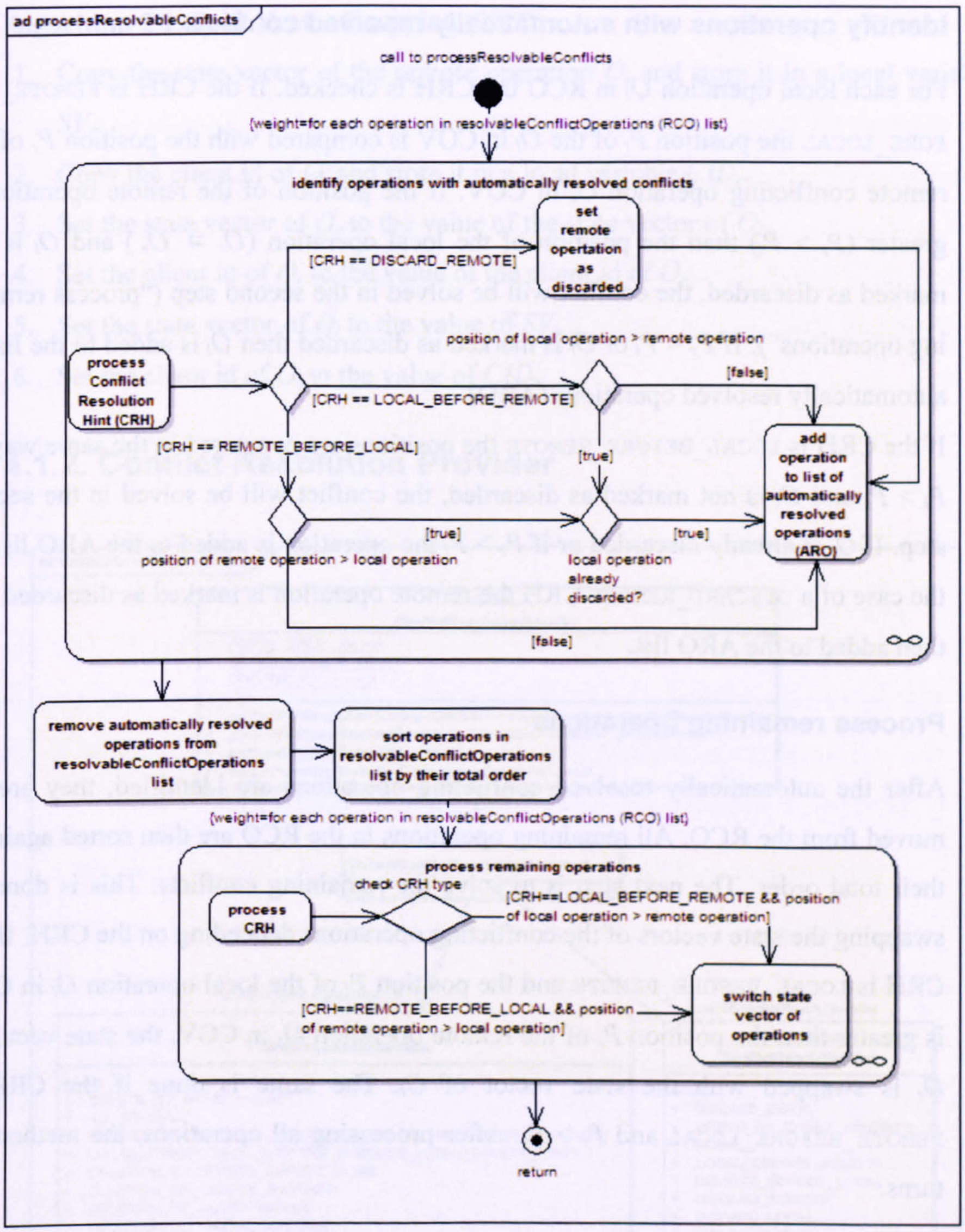

Figure 4.9: Processing resolvable conflicts activity diagram

Discarded operation can be marked as not discarded again if, for example, the operation that lead to the discarding of another operation is undone.

Identify operations with automatically resolved conflicts

For each local operation O_l in RCO the CRH is checked. If the CRH is `REMOTE_BEFORE_LOCAL` the position P_l of the O_l in COV is compared with the position P_r of the remote conflicting operation O_r in COV. If the position of the remote operation is greater ($P_r > P_l$) than the position of the local operation ($O_l \Rightarrow O_r$) and O_l is not marked as discarded, the conflict will be solved in the second step ("process remaining operations"). If $P_r < P_l$ or O_l is marked as discarded then O_l is added to the list of automatically resolved operations (ARO).

If the CRH is `LOCAL_BEFORE_REMOTE` the positions are compared in the same way. If $P_l > P_r$ and O_l is not marked as discarded, the conflict will be solved in the second step. If O_l is already discarded or if $P_r > P_l$ the operation is added to the ARO list. In the case of a `DISCARD_REMOTE` CRH the remote operation is marked as discarded and then added to the ARO list.

Process remaining operations

After the automatically resolved conflicting operations are identified, they are removed from the RCO. All remaining operations in the RCO are then sorted again by their total order. The next step is to solve the remaining conflicts. This is done by swapping the state vectors of the conflicting operations depending on the CRH. If the CRH is `LOCAL_BEFORE_REMOTE` and the position P_l of the local operation O_l in COV is greater than the position P_r of the remote operation O_r in COV, the state vector of O_r is swapped with the state vector of O_l. The same is done if the CRH is `REMOTE_BEFORE_LOCAL` and $P_r > P_l$. After processing all operations, the method returns.

Swapping state vectors of operations

The swapping of the state vectors is done in order to change the total ordering position of an operation within the COV. In the case of two concurrent operations with identical state vector values, the client id is the determining factor for the total ordering position of an operation. Thus, together with the state vector the client ids are swapped. Algorithm 2.1 specifies how the state vector swapping is implemented.

Algorithm 2.1. State Vector Swapping (SVS)

1. Copy the state vector of the remote operation O_r and store it in a local variable SV_r.
2. Copy the client id of O_r and store it in a local variable CID_r.
3. Set the state vector of O_r to the value of the state vector of O_l.
4. Set the client id of O_r to the value of the client id of O_l.
5. Set the state vector of O_l to the value of SV_r.
6. Set the client id of O_l to the value of CID_r.

4.1.2. Conflict Resolution Provider

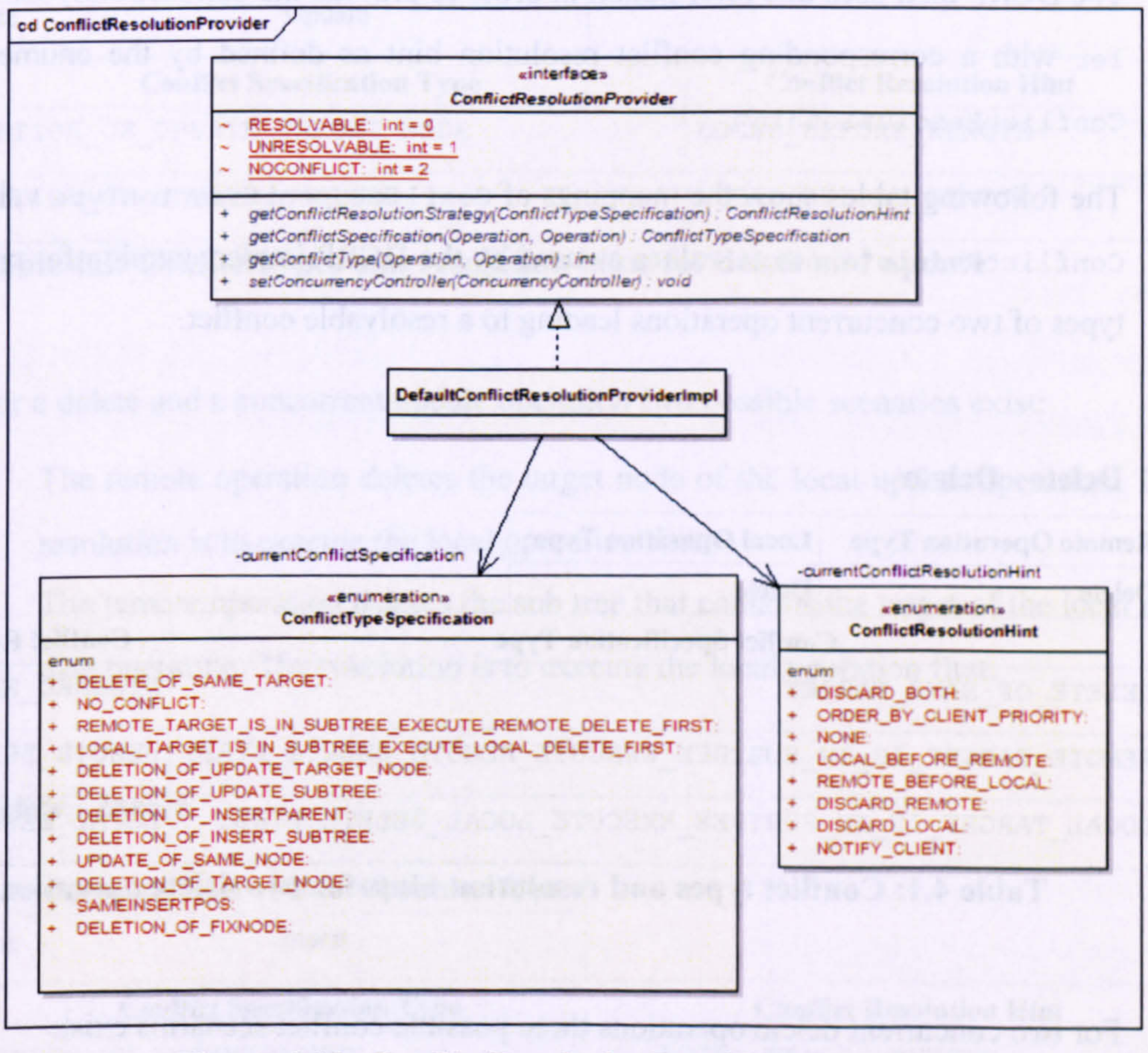

Figure 4.10: ConflictResolutionProvider class diagram

The job of the conflict resolution provider is to identify a conflict and provide the `ConcurrencyController` with a `ConflictResolutionHint` defining an action that

is to be taken in order to resolve the conflict. The class diagram in figure 4.10 shows the relations between the `ConflictResolutionProvider` interface and its implementing class, the `DefaultConflictResolutionProviderImpl`. The `DefaultConflictResolutionProviderImpl` (DCRP) class has methods for checking operations of a specific type (delete, insert or update). They return a specification of a conflict as defined by the `ConflictTypeSpecification` enumeration. For example, the method `checkInsertDelete(InsertOperation insert, DeleteOperation delete)` is called if the remote operation is a delete operation and the local operation is an insert operation. The two concurrent operations are then checked for a conflict and the information on the type of conflict, if any, is returned. The DCRP then uses this information in order to provide the `ConcurrencyController` with a corresponding conflict resolution hint as defined by the enumeration `ConflictResolutionHint`.

The following tables show the mappings of `ConflictSpecificationType` values to `ConflictResolutionHint` values as used by the DCRP implementation for specific types of two concurrent operations leading to a resolvable conflict.

Delete - Delete

Remote Operation Type	**Local Operation Type**
Delete	Delete

Conflict Specification Type	**Conflict Resolution Hint**
DELETE_OF_SAME_TARGET	DISCARD_REMOTE
REMOTE_TARGET_IS_IN_SUBTREE_EXECUTE_REMOTE_DELETE_FIRST	REMOTE_BEFORE_LOCAL
LOCAL_TARGET_IS_IN_SUBTREE_EXECUTE_LOCAL_DELETE_FIRST	LOCAL_BEFORE_REMOTE

Table 4.1: Conflict types and resolution hints for two delete operations

For two concurrent delete operations three possible conflict scenarios exist:

- Both operations delete the same target. The resolution is to discard the remote operation. Both operations have the same effect when executed, they delete the same target node. Thus it is not important which of the two operations is discarded. Discarding the remote operations is an arbitrary decision. Alternatively,

it would be possible for example, to discard the operation with the greater position value in the COV.

- The target node of the remote operation is part of the sub tree that is to be deleted by the local operation. The resolution is to execute the remote operation first.
- The target node of the local operation is part of the sub tree that is to be deleted by the remote operation. The resolution is to execute the local operation first.

Delete - Update

Remote Operation Type	Local Operation Type
Delete	Update

Conflict Specification Type	Conflict Resolution Hint
DELETION_OF_UPDATE_TARGET_NODE	LOCAL_BEFORE_REMOTE
DELETION_OF_UPDATE_SUBTREE	LOCAL_BEFORE_REMOTE

Table 4.2: Conflict types and resolution hints for delete and update

For a delete and a concurrent update operation two possible scenarios exist:

- The remote operation deletes the target node of the local update operation. The resolution is to execute the local operation first.
- The remote operation deletes the sub tree that contains the target of the local update operation. The resolution is to execute the local operation first.

Delete - Insert

Remote Operation Type	Local Operation Type
Delete	Insert

Conflict Specification Type	Conflict Resolution Hint
DELETION_OF_INSERTPARENT	LOCAL_BEFORE_REMOTE
DELETION_OF_FIXNODE	LOCAL_BEFORE_REMOTE
DELETION_OF_INSERT_SUBTREE	LOCAL_BEFORE_REMOTE

Table 4.3: Conflict types and resolution hints for delete and insert

For a delete and a concurrent insert operation the following conflict scenarios exist:

- The remote operation deletes the parent node of the node that is to be inserted by the local insert operation. The resolution is to execute the local operation first.
- The remote operation deletes the fix node of the node that is to be inserted by the local insert operation. The resolution is to execute the local operation first.
- The remote operation deletes the sub tree that will contain the node that is to be inserted by the local insert operation. The resolution is to execute the local operation first.

Update - Delete

Remote Operation Type	Local Operation Type
Update	Delete

Conflict Specification Type	Conflict Resolution Hint
DELETION_OF_TARGET_NODE	REMOTE_BEFORE_LOCAL
DELETION_OF_UPDATE_SUBTREE	REMOTE_BEFORE_LOCAL

Table 4.4: Conflict types and resolution hints for update and delete

For a remote update and a concurrent local delete operation the following conflict scenarios exist:

- The local operation deletes the target node of the remote update operation. The resolution is to execute the remote operation first.
- The local operation deletes the sub tree that contains the target node of the remote update operation. The resolution is to execute the remote operation first.

Insert - Delete

Remote Operation Type	Local Operation Type
Insert	Delete

Conflict Specification Type	Conflict Resolution Hint
DELETION_OF_FIXNODE	REMOTE_BEFORE_LOCAL
DELETION_OF_INSERTPARENT	REMOTE_BEFORE_LOCAL
DELETION_OF_INSERT_SUBTREE	REMOTE_BEFORE_LOCAL

Table 4.5: Conflict types and resolution hints for insert and delete

For a remote insert operation and a concurrent local delete operation the following conflict scenarios exist:

- The local operation deletes the fix node of the insert operation. The resolution is to execute the remote operation first.
- The local operation deletes the parent of the node that is to be inserted by the remote operation. The resolution is to execute the remote operation first.
- The local operation deletes the sub tree that will contain the node that is to be inserted by the remote operation. The resolution is to execute the remote operation first.

If no conflict is found, the `ConflictResolutionProvider` will return the `ConflictTypeSpecification` value `NO_CONFLICT` for a pair of concurrent operations. In this case the CRH value is set to `NONE`. Other possible CRH values, next to the ones described above, are:

- `DISCARD_BOTH`. This indicates to mark both concurrent operations as discarded.
- `DISCARD_LOCAL`. This indicates to mark the local operation as discarded.
- `ORDER_BY_CLIENT_PRIORITY`. This indicates to order the operations by client priority instead of their total order.
- `NOTIFY_CLIENT`. This indicates to notify the client of the conflict in order to retrieve a CRH from the client application. This may be used for cases where the conflict should be solved by a user action.

These CRH values are not used in this implementation of the `ConflictResolutionProvider` but were defined so that they can be used by third party implementers of the `ConflictResolutionProvider` interface.

4.1.3. Operations

The basic set of operations used in this concurrency control algorithm includes insert, delete and update. Moving of nodes is achieved by delete and subsequent insert operations at the application level. For moving a whole sub tree of a document only one delete and one insert operation is necessary keeping a move operation affordable in terms of application performance. An operation is executed on an element node N ($N \in D$). Element nodes contain a map of attribute nodes AM_N and an ordered list of child nodes CL_N[41]. Element nodes are identified by the above-mentioned UUID. D denotes the XML document tree containing all nodes of the current document as specified by the W3C.

The used operations are defined as follows:

- `INSERT(N,LOC(X,REL(C,0)),SV,CID)`
- `DELETE(N,SV,CID)`
- `UPDATE(N,NMOD(AM,TL),SV,CID)`

LOC(X,REL(C,0)) denotes a location within the XML document. The location is specified by the target node *X* that will become parent of the new node and the relative position (*REL*) to an existing child element node *C*. *0* denotes the relative location before *C*, *1* denotes the relative location after *C*. If the target node does not contain any child element nodes, the new node is simply appended. *SV* denotes the state vector of the generating client site, *CID* the identifier of the generating client site. All sites that are part of a concurrent editing session receive the *CID* from the session server, when they join the session. *NMOD* denotes the modification set that is applied on the node *N*. A modification set contains a map of modified attributes *AM* and a list of modified texts *TL*. An attribute *A* ($A \in AM$) existing in *N* ($A \in AM_N$) is updated to the new value, a new attribute ($A \notin AM_N$) is added to the corresponding node and an attribute $A \notin AM$ and $A \in AM_N$ is removed from AM_N. Figure 4.11 shows the Operation interfaces and implementing classes.

---

[41] See the Document Object Model specification for the definition of DOM element nodes at: http://www.w3.org/DOM/ , retrieved October 30, 2007

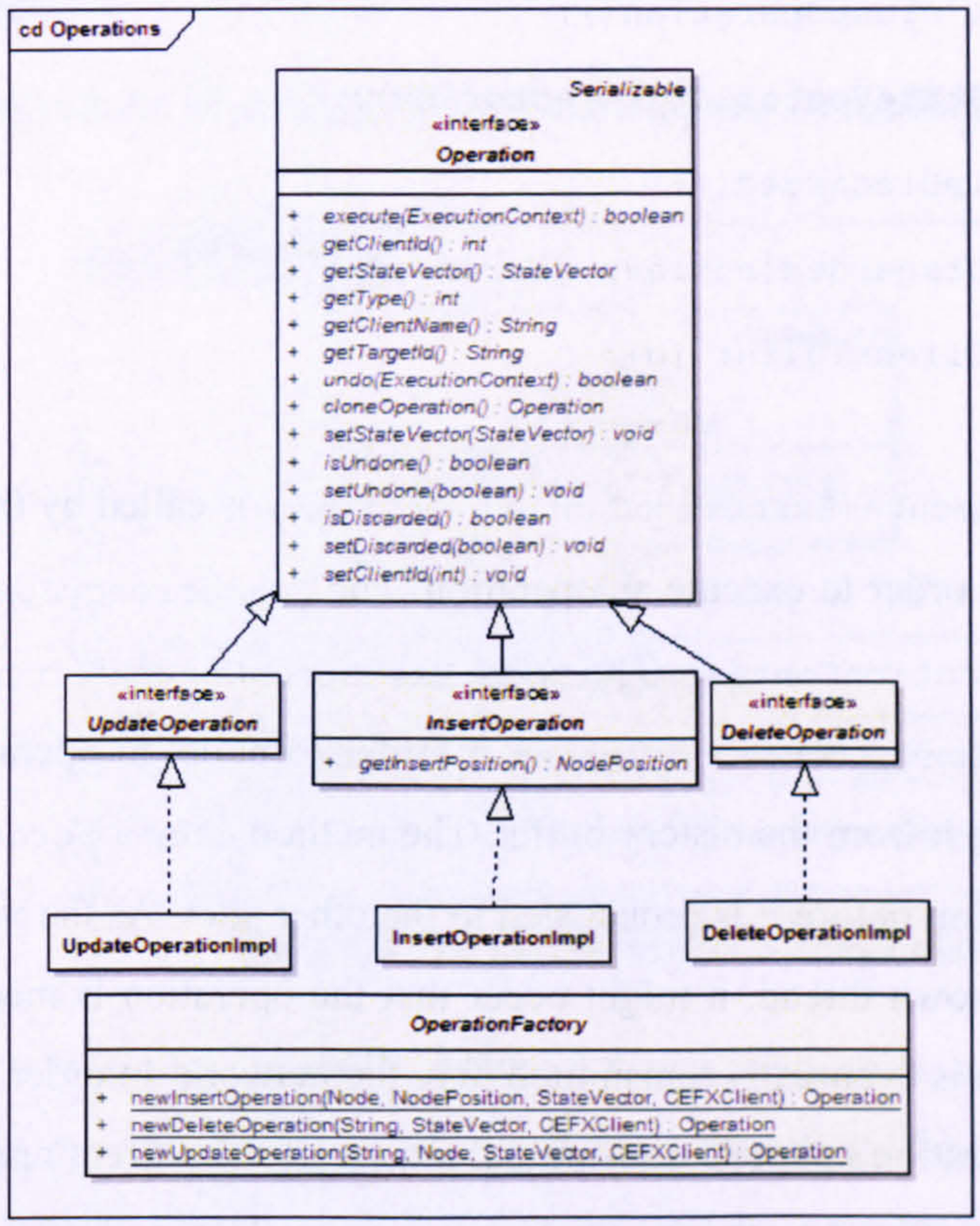

Figure 4.11: Operation interfaces and classes

The `Operation` interface is the main interface for all three types of operations and defines the methods that all operations have in common. An additional interface exists for each operation type (update, insert and delete). These interfaces inherit the methods from the `Operation` interface. The classes `UpdateOperationImpl`, `InsertOperationImpl` and `DeleteOperationImpl` implement those interfaces. The `OperationFactory` is a utility class for the instantiation of operation objects.

The `Operation` interface defines the following methods:

- `boolean execute(ExecutionContext context);`
- `int getClientId();`
- `StateVector getStateVector();`
- `int getType();`
- `String getClientName();`
- `String getTargetId();`
- `boolean undo(ExecutionContext context);`

- `Operation cloneOperation();`
- `void setStateVector(StateVector sv);`
- `boolean isDiscarded();`
- `void setDiscarded(boolean discarded);`
- `void setClientId(int id);`

The method `execute(ExecutionContext context)` is called by the `Concurrency-Controller` in order to execute an operation. The `ConcurrencyController` thereby acts as the `ExecutionContext`. The `undo(ExecutionContext context)` method is called by the `ConcurrencyController` in order to undo an operation for example when removing it from the history buffer. The method `cloneOperation()` is used to copy an operation before it is propagated to the other sites. As the `NetworkControl-ler` runs in its own thread, it might occur that the operation is modified by another thread before it is eventually transmitted over the network. In order to make sure that the correct operation's state is transmitted, a copy of the current operation's values is taken before passing it to the `NetworkController`. The method `getType()` returns the type (delete, insert or update) of the operation. This is used, for example in the DCRP in order to easily identify the type of an operation. The method `getClient-Name()` returns the name of the client that created the operation. This is used for debugging purposes and in order to be able to later identify the operation's originating client if the client id has been swapped. The other getter and setter methods are used as their names indicate for retrieving or setting property values of the operation.

The following sections discuss the three different operation implementation classes.

4.1.3.1. Insert

Figure 4.12 shows the `InsertOperation` interface and its methods.

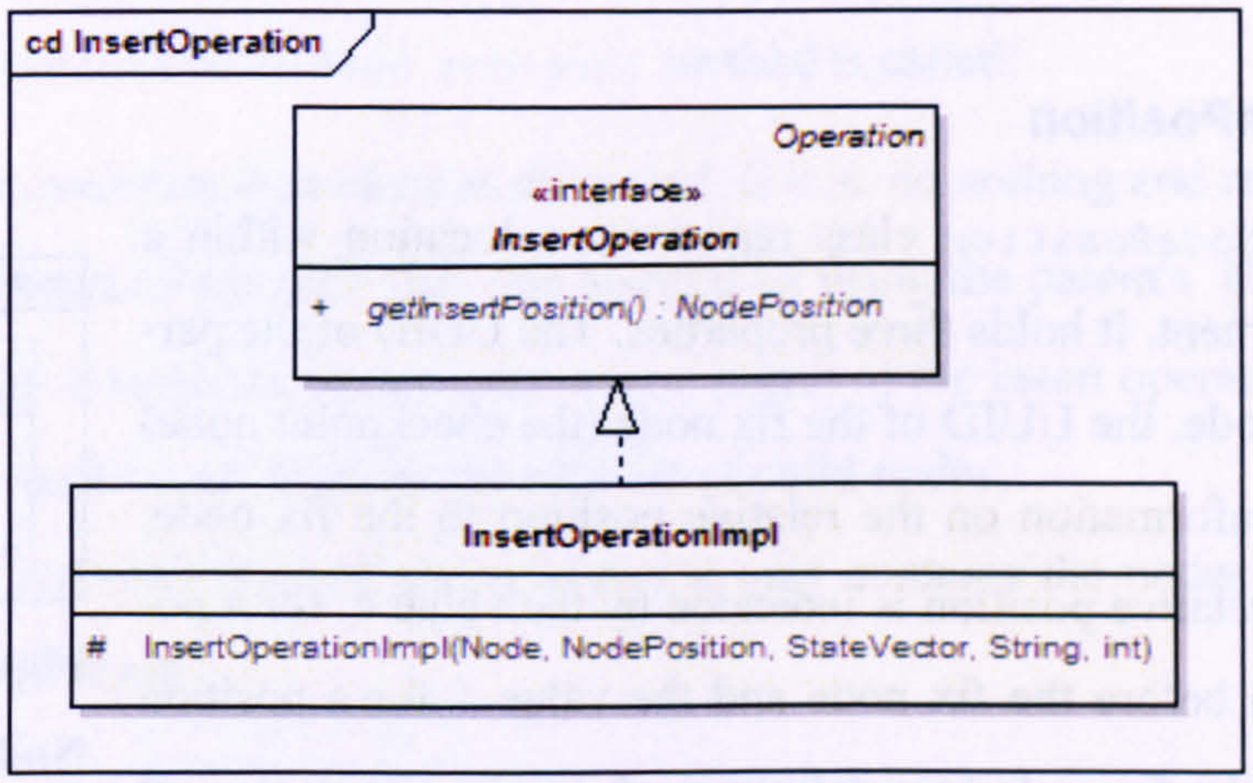

Figure 4.12: InsertOperationImpl class

The `InsertOperation` interface defines the `getInsertPosition()` method. All other methods are inherited from the `Operation` interface. The `InsertOperation-Impl` class implements `InsertOperation` and defines a constructor that requires the following arguments in order to instantiate a `InsertOperationImpl` object:

- `Node` – a reference to the `org.w3c.dom.Node` object that is to be inserted in the document.
- `NodePosition` – a reference to the `NodePosition` object that defines the location where to insert the new node.
- `StateVector` – a reference to the `StateVector` containing the initial state vector values for this operation.
- `String` – the name of the client that creates this operation as `String`.
- `int` – the identifier of the client that creates this operation.

Node

The node which is inserted when executing an insert operation must implement the standard `org.w3c.dom.Node` interface object as defined by the World Wide Web Consortium. When the operation is transmitted over the network, it is serialized into a byte stream. The node that is contained in the operation must therefore also imple-

ment the `java.io.Serializable` interface. This requirement is fulfilled for example by the Apache Xerces DOM implementation which is used in this works implementation.

NodePosition

The `NodePosition` class represents a location within a document. It holds three properties: The UUID of the parent node, the UUID of the fix node (the checkpoint node) and information on the relative position to the fix node. The relative position is indicated by the value `0` for a position before the fix node and the value `1` for a position after the fix node (see definition of *`LOC(X,REL(C,0))`* at the beginning of this chapter).

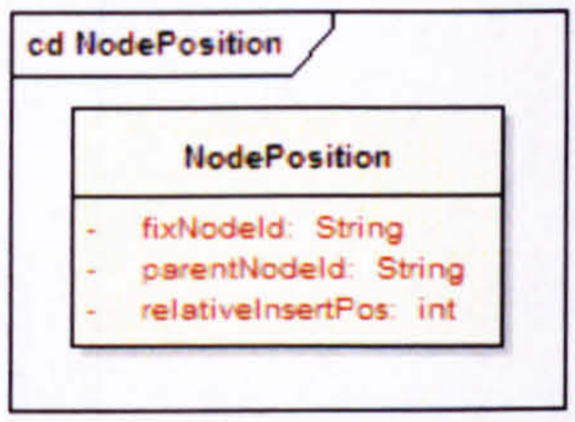

Figure 4.13: Class NodePosition

Executing an insert operation

When the `execute(ExecutionContext context)` method of the `InsertOperationImpl` class is called, the following steps are executed:

- Refresh the context's map of nodes by calling `context.refreshNodeMap()`. This is done in order to make sure that all existing nodes within the document can be found easily.
- Check if the operation is marked as discarded and if it is, do nothing and return.
- Insert the new node at the given position (`NodePosition`) within the document. This is done by using the standard DOM interface methods to manipulate the document. For detailed information on the implementation, please refer to the thesis' attached source code.
- If the node was successfully inserted, the context's map of nodes is refreshed again.

Undoing an insert operation

The `undo(ExecutionContext context)` method is used to recreate the document's state before the execution of the insert method. The following steps are executed when the `undo(ExecutionContext context)` method is called:

- Check if the operation is marked as discarded. If it is, do nothing and return.
- Locate the parent of the node that was inserted by using the parent's UUID. The parents UUID is stored in the `NodePosition` object of the insert operation.
- Remove the target node from its parent's list of child nodes.
- Refresh the context's map of nodes so that it only contains the nodes that exist within the document.

4.1.3.2. Delete

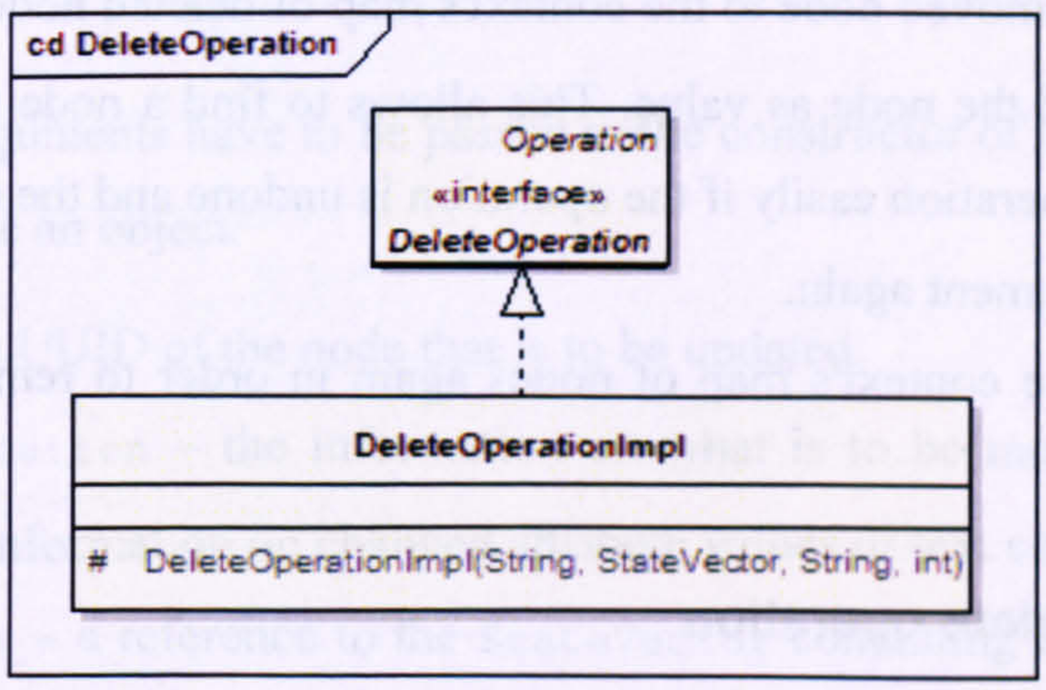

Figure 4.14: DeleteOperationImpl class

Figure 4.14 shows the `DeleteOperation` interface and the `DeleteOperationImpl` class.

The `DeleteOperationImpl` class implements the `DeleteOperation` interface and requires the following arguments to be passed to the constructor in order to instantiate it:

- `String` – the UUID of the node that is to be deleted.
- `StateVector` – a reference to the `StateVector` containing the initial state vector values for this operation.
- `String` – the name of the client that creates this operation as `String`.

- `int` – the identifier of the client that creates this operation.

Executing a delete operation

When the `execute(ExecutionContext context)` of the `DeleteOperationImpl` class is called, the following steps are executed:

- Refresh the context's map of nodes.
- Check if the operation is marked as discarded and if it is, do nothing and return.
- Find the node that is to be deleted (the target node) by using the given UUID.
- Retrieve the parent node of the target node and memorize a reference to it for a later undo.
- Memorize the relative position of the node that is to be deleted by storing a reference to node next to the target node.
- Remove the target node from the parent node's list of child nodes.
- Add the removed node to the context's map of deleted nodes using the operation as key and the node as value. This allows to find a node that was deleted by a specific operation easily if the operation is undone and the node must be inserted in the document again.
- Refresh the context's map of nodes again in order to remove the deleted node from it.

Undoing a delete operation

The following steps are executed when the `undo(ExecutionContext context)` operation is called on the `DeleteOperationImpl` class:

- Check if the operation is marked as discarded and if it is, do nothing and return.
- Retrieve the deleted node from the context's map of deleted nodes.
- Insert the node at the memorized position below its parent node.
- Refresh the context's map of nodes in order to include the inserted node again.

4.1.3.3. Update

Figure 4.14 shows the `UpdateOperation` interface and its implementing class `UpdateOperationImpl`.

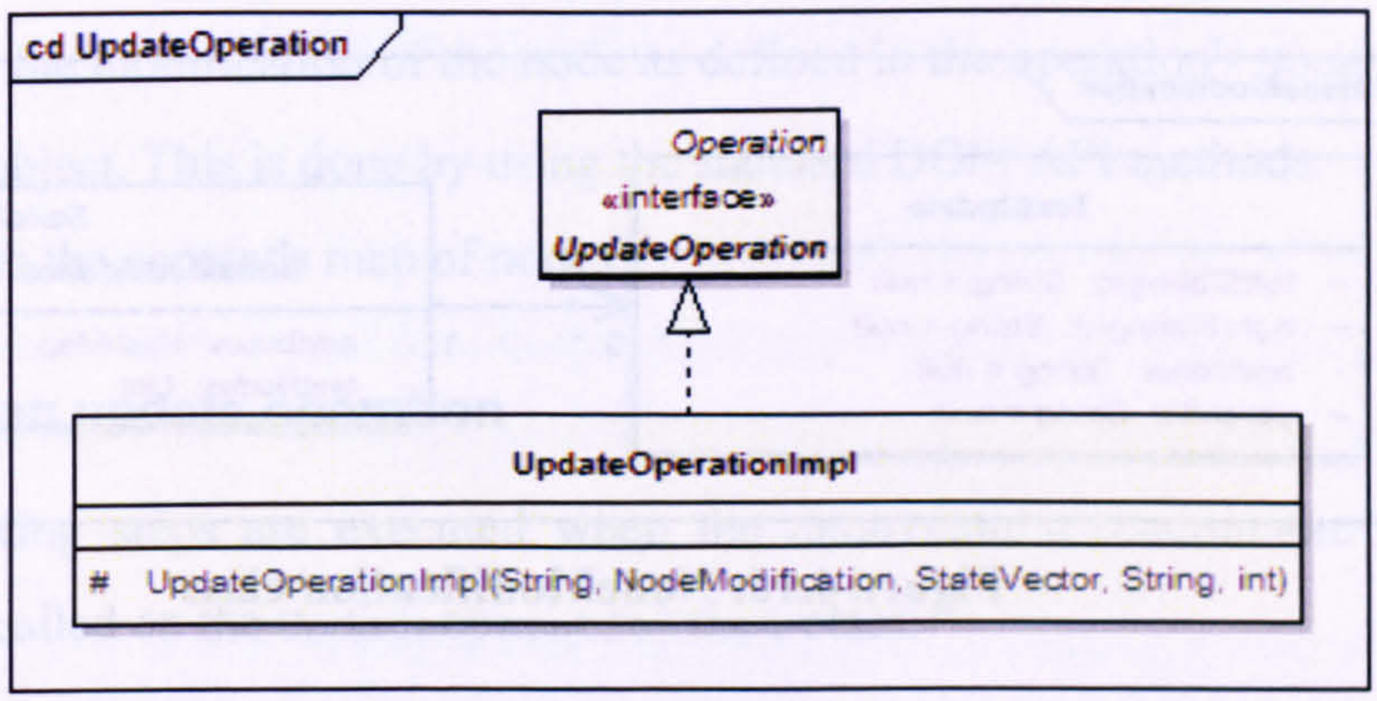

Figure 4.15: UpdateOperationImpl class

The following arguments have to be passed to the constructor of `UpdateOperationImpl` to instantiate an object:

- `String` – the UUID of the node that is to be updated.
- `NodeModification` – the information on what is to be modified. This object contains all information on changed attribute values or text content values.
- `StateVector` – a reference to the `StateVector` containing the initial state vector values for this operation.
- `String` – the name of the client that creates this operation as `String`.
- `int` – the identifier of the client that creates this operation.

NodeModification

The `NodeModification` class was designed for two reasons:

1. To provide a simple method for storing the modified state of a node object. A node thereby can be modified using the standard DOM API methods. After modifying the node, it is passed as argument to the constructor of the `NodeModification` class which then stores all changes in the new `NodeModification`

object.

2. To provide a container format to transmit these modifications over the network without having to transmit a complete node including possible child element nodes. This allows reduction of network traffic when executing update operations.

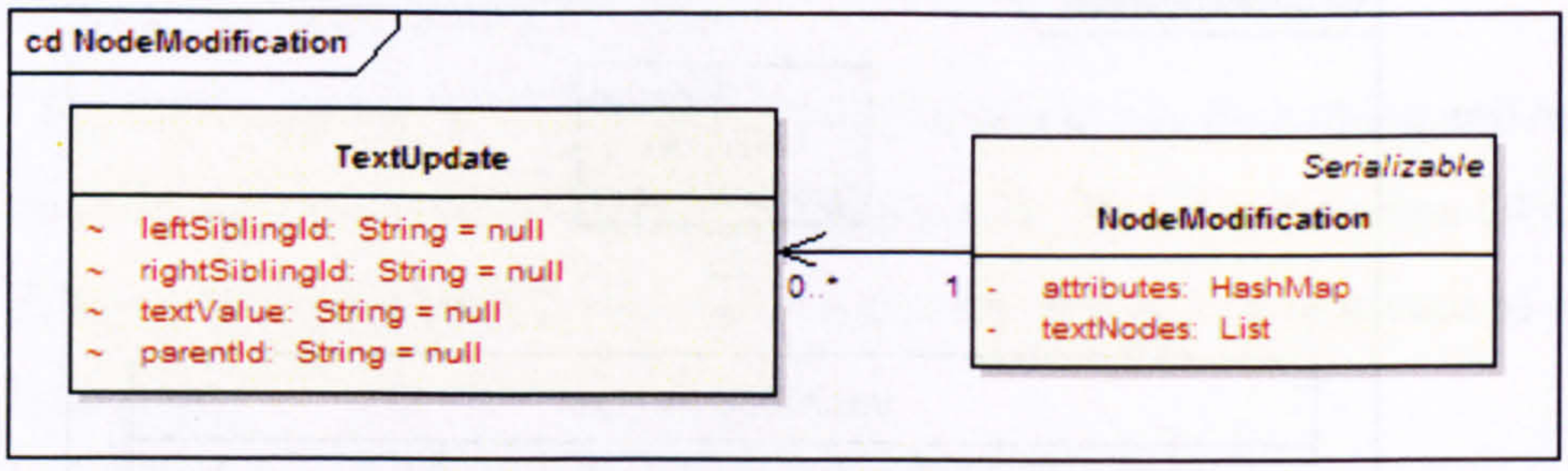

Figure 4.16: NodeModification class

A `NodeModification` object stores the attributes and the text content of a node. The attributes are stored in a mapping table containing all attribute names and their corresponding attribute values.

The W3C DOM API specifies, that the text content of an element node[42] is stored in a `Node` object of the type `TEXT_NODE`. These specialised *text nodes* only carry a text as content and have no child nodes. Within a `NodeModification` object the text content of an element node (the content of all of its text nodes) is stored in a list containing objects of the type `TextUpdate`. A `TextUpdate` contains the UUID of a text node's parent node (the target node), the UUIDs of its left and right siblings (if existing) and the text content.

Figure 4.16 shows the properties of the `TextUpdate` and `NodeModification` classes and their connection.

Executing an update operation

When the `execute(ExecutionContext context)` of the `UpdateOperationImpl` class is called, the following steps are executed:

- Refresh the context's map of nodes.

---

[42] Element nodes can carry attributes, text content and child nodes.

- Check if the operation is marked as discarded and if it is, do nothing and return.
- Locate the target node in the context's document by using the UUID.
- Normalize the target node's text content[43].
- Store the current state of the target node in a `NodeModification` object in order to be able to do a later undo of the operation.
- Apply the modification of the node as defined in the operation's `NodeModification` object. This is done by using the standard DOM API methods.
- Refresh the context's map of nodes.

Undoing an update operation

The following steps are executed when the `undo(ExecutionContext context)` method is called on the `UpdateOperationImpl` class:

- Check if the operation is marked as discarded and if it is, do nothing and return.
- Refresh the context's map of nodes.
- Locate the target node in the context's document by using the UUID.
- Normalize the target node's text content.
- Apply the modification of the target node as defined in the previously stored `NodeModification` object to restore the old node state.
- Refresh the context's map of nodes.

4.2. Testing CMAX

A simulation software was developed prior to the development of the `ConcurrencyController` implementation. The simulation software was used in order to test the CMAX software implementation during the development process.

The simulation algorithm simulates an arbitrary number of users (usually three users are chosen) working concurrently on a shared XML document. It consists of a client component part and a server component part. The client component simulates the editing application and the user's actions that lead to a modification of the shared document. The server component handles the session management.

---

[43] Normalizing a node makes sure that there are neither adjacent text nodes nor empty text nodes. See the W3C DOM API specification for a detailed explanation of the normalize function.

When the simulation is started each client connects itself to the server and retrieves a copy of the shared XML document (the simulation uses an XML DocBook[44] document). When all clients are connected the editing session is initialised and each client concurrently executes random operations on the shared document. After a certain time the process is stopped and each client's and the server's document is stored. The test was deemed successful if all documents were identical.

In order to test the algorithm on certain concurrent operations that are known to be problematic, the simulation algorithm was extended. With that extension the simulation algorithm was provided with a list of operations that each client has to execute at a certain time. This allowed testing the software on both certain problematic operation scenarios or on random operations.

This simulation procedure helped to reduce development time and increase the stability of the CMAX software implementation. Without this simulation procedure it would have been more difficult to test cases where, for example, three users concurrently access the same node of a document.

The client and the server component use the `OrderingConcurrencyControllerImpl` implementation discussed above for the synchronisation of the XML document. The following section briefly discusses the implementation of the simulation procedure as a practical Java software implementation.

4.2.1. Simulation software implementation

Figure 4.17 shows an overview of the client and server components of the simulation software and their connections to each other and the concurrency controller implementation. The client is represented by the `Client` interface which is implemented by the `SimulationClient` class. The server has a reference to each client that is connected to the session. The server is represented by the `Server` interface which is implemented by the `SimulationServer` class. A client knows the server (through the `Server` interface), as the server provides the client with a copy of the shared document. Each client and the server own a `ConcurrencyController` object.

When a client or a server is initialised, an object of the type `OrderingConcurrency-`

[44] XML DocBook is an XML based markup language for technical documentation.

ControllerImpl is initialised too and is provided with a reference to an OperationExecutor.

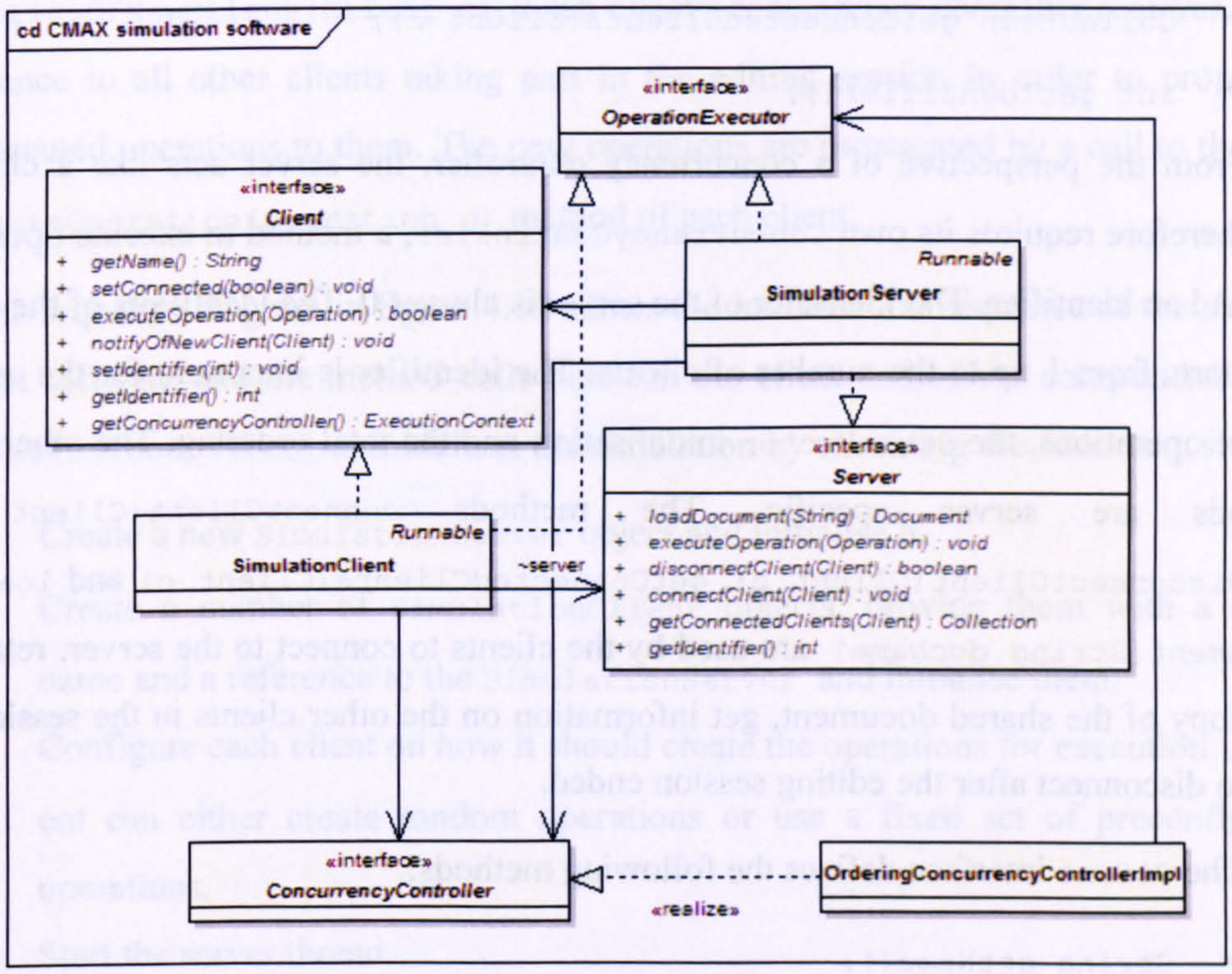

Figure 4.17: Simulation software server and client components

As shown in the class diagram in figure 4.17 the OperationExecutor interface is implemented by the client and the server. The SimulationServer and SimulationClient classes are implemented as threads (implementing the java.lang.Runnable interface). They run independent of each other and communicate only via the corresponding interfaces. This allows each client to execute operations at any time, independent of the other clients or the server. Although, the implementation effort of the simulation software increased because of the thread safety issues that needed to be tackled, the benefits of the multi-threaded design of the simulation software were worth the implementation effort.

The Server interface defines the following methods:

- `Document loadDocument(String docName);`
- `void executeOperation(Operation o);`

- `boolean disconnectClient(Client c);`
- `void connectClient(Client c);`
- `Collection getConnectedClients(Client c);`
- `int getIdentifier();`

From the perspective of a concurrency controller, the server acts like a client. It therefore requires its own `ConcurrencyController`, a method to execute operations and an identifier. The identifier of the server is always 0. The identifiers of the clients starts from 1 up to the number of clients. The identifier is important for the creation of operations, the `StateVector` initialisation and the total ordering. The other methods are server specific. The methods `connectClient(Client c)`, `disconnectClient(Client c)`, `getConnectedClients(Client c)` and `loadDocument(String docName)` are used by the clients to connect to the server, retrieve a copy of the shared document, get information on the other clients in the session and to disconnect after the editing session ended.

The `Client` interface defines the following methods:

- `String getName();`
- `void setConnected(boolean c);`
- `boolean executeOperation(Operation o);`
- `void notifyOfNewClient(Client c);`
- `void setIdentifier(int id);`
- `int getIdentifier();`
- `ExecutionContext getConcurrencyController();`

Each client has a name and an identifier. The methods `getName()` and `getIdentifier()` are used to retrieve those client properties. Additionally an operation is provided with a reference to the `ExecutionContext` of a client, represented by the clients concurrency controller implementation (see chapter 4.1.3). In order to retrieve the `ExecutionContext` of a client when creating an operation the `getConcurrencyController()` method of the `Client` interface is used. The identifier of a client is generated by the server when the client connects to it. The server uses the method `setIdentifier(int id)` to set a client's identifier. When a client is successfully connected to the server, the server uses the `setConnected(boolean c)` method to

notify the client that it is connected to the editing session. The server subsequently notifies all other connected clients of the new client connection by calling the method `notifyOfNewClient(Client c)`. Each client's concurrency controller requires a reference to all other clients taking part in the editing session in order to propagate executed operations to them. The new operations are propagated by a call to the `executeOperation(Operation o)` method of each client.

The sequence diagram in figure 4.18 shows the initialisation sequence of the simulation software and the method calls between the client and server components. The `SimulationController` initialises the simulation by executing the following steps:

- Create a new `SimulationServer` object and initialise it.
- Create a number of `SimulationClient` objects, provide them with a client name and a reference to the `SimulationServer` and initialise them.
- Configure each client on how it should create the operations for execution. A client can either create random operations or use a fixed set of preconfigured operations.
- Start the server thread.
- Start the client threads.
- Wait for a certain amount of time.
- Stop the client threads. Each client disconnects from the server by calling the `Server` interface method `disconnectClient(Client c)` before it stops.
- Stop the server thread.

The initialisation of the `SimulationServer` object includes the following steps as shown in figure 4.18:

- Create an `OrderingConcurrencyControllerImpl` object and provide it with a reference to the `ExecutionContext` (represented by the server).
- Load the XML document into memory.
- Create the UUIDs for each node in the document and create a map of all nodes in the document.
- Provide the `ConcurrencyController` with a reference to the document by calling `setDocument(Document localDoc)`.

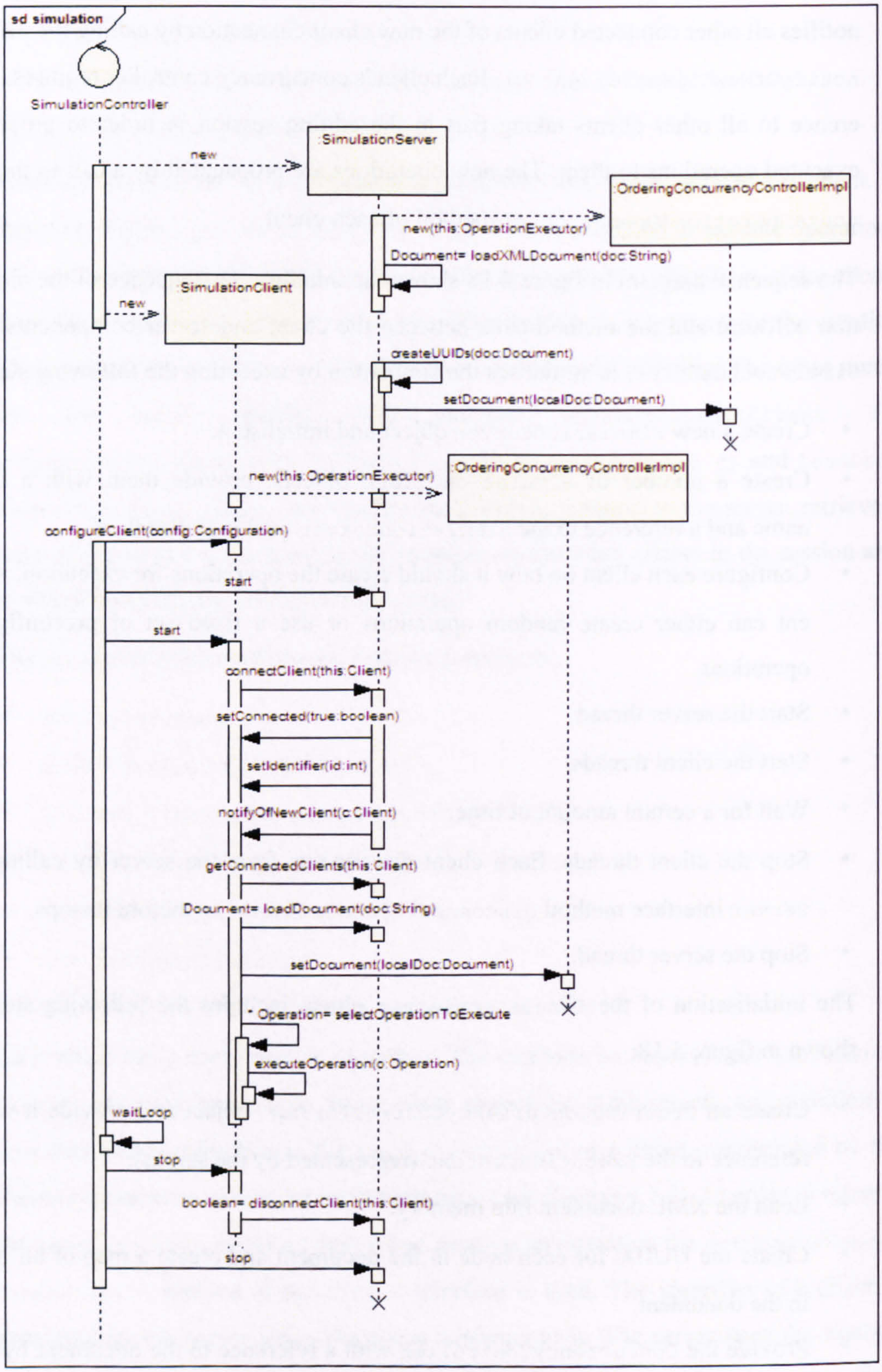

Figure 4.18: Simulation sequence diagram

When the `SimulationClient` object is initialised it creates an `OrderingConcurrencyControllerImpl` object and provides it with a reference to the `ExecutionContext` (represented by the client). After a client is started the following steps are executed:

- Connect the client with the server. The client calls the `connectClient(Client c)` method of the `Server` interface and provides the server with a reference to the client.
- The server calls the client's `setConnected(boolean c)` method and thereby notifies the client that it is connected to the server.
- The server creates an identifier for the client and provides the client with it by calling the `setIdentifier(int id)` method of the `Client` interface.
- The server notifies each client in the session of the new connected client and provides them with a reference to the new client object by calling the method `notifyOfNewClient(Client c)`.
- The client calls the method `getConnectedClients(Client c)` of the `Server` interface in order to retrieve a complete list of all clients in the session. This is done to make sure that all clients have a complete list of all other clients in the session. Clients that join the session late use this method to get references to the clients that joined the session earlier.
- The client retrieves the shared document from the server by calling the `loadDocument(String doc)` method of the `Server` interface.
- The client starts editing the document in a loop, selects different operations and executes them. The `ConcurrencyController` of each client executes the operation locally and propagates it to the other clients as discussed in chapter 4.1.1 (not shown in the sequence diagram in figure 4.18).

4.3. Integration of CMAX in CEFX

The next step after implementing the CMAX algorithm in software was to integrate the developed software components into the Collaborative Editing Framework for XML (CEFX). CEFX consists of a number of loosely coupled components where each component can be replaced by a different implementation.

This simplified the integration of CMAX. The CMAX implementation basically consists of a number of classes including the concurrency controller, the conflict resolution provider and the operations. In order to integrate those classes into the CEFX default implementation, they are arranged into two packages. The first package is called the `concurrency` package. The second package, which is part of the `concurrency` package, is called `operations`. Within the CEFX package structure these packages are arranged as shown in figure 4.19. The `concurrency` package contains all classes that are responsible for maintaining the consistency of a shared document. The main component here is the `ConcurrencyController` as described in chapter 4.1. Within CEFX the `CEFXController` binds the `ConcurrencyController` to the framework and delegates all events concerning the execution of operations to it. The following interfaces and classes are part of the concurrency package:

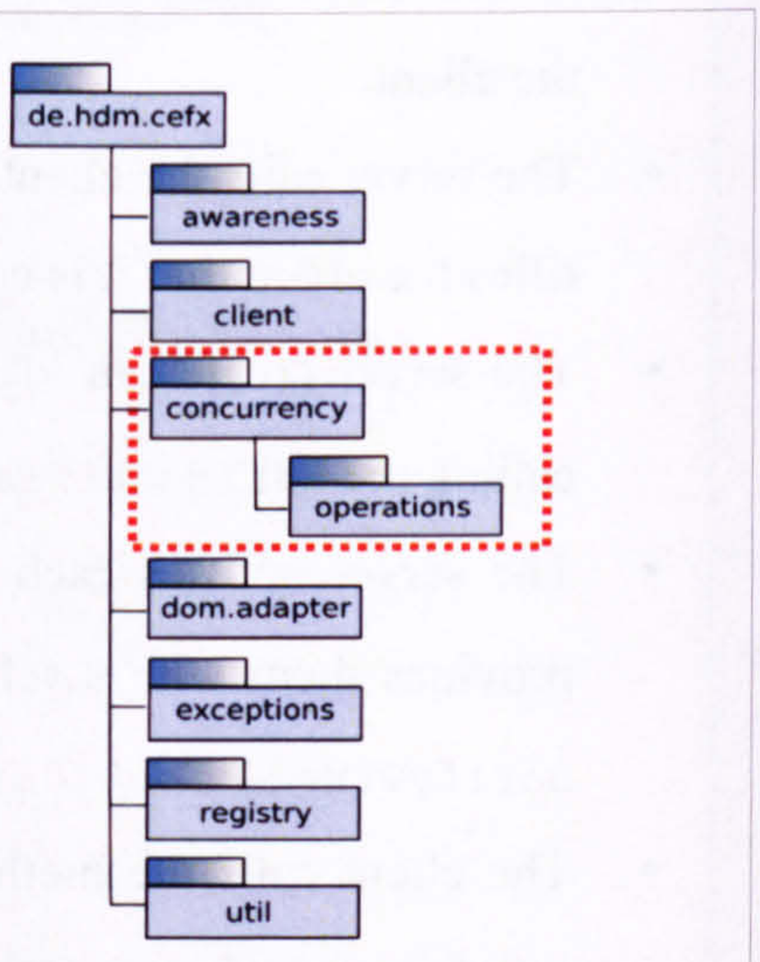

Figure 4.19: CEFX package structure

- The `ConcurrencyController` interface and its implementing classes `AbstractConcurrencyControllerImpl` and `OrderingConcurrencyControllerImpl`.
- The `ConflictResolutionProvider` interface and its implementing class `DefaultConflictResolutionProvider`.
- The `ConflictTypeSpecification` and `ConflictResolutionHint` enumeration.

- Utility classes such as the `ConflictInfo` and `ConflictList` class.

The `operations` package contains all classes and interfaces that model operations and are necessary for the execution of an operation. This package contains the following interfaces and classes:

- The `Operation` interface and the sub-interfaces `DeleteOperation`, `InsertOperation`, `UpdateOperation` and the implementation classes of operations: `DeleteOperationImpl`, `InsertOperationImpl` and `UpdateOperationImpl`.
- The `OperationFactory` class for creating operations and other utility classes.
- The `StateVector` class.
- The `ExecutionContext` interface which is implemented by the `ConcurrencyController` implementation.
- The `OperationExecutor` interface which is implemented by the `CEFXController`.
- The `NodePosition` class for identification of a node's location within a document and the `NodeModification` and `TextUpdate` classes for the update operations.

The class diagram in figure 4.20 shows an overview of the packages, classes, interfaces and their correlation used for the software implementation of CMAX.

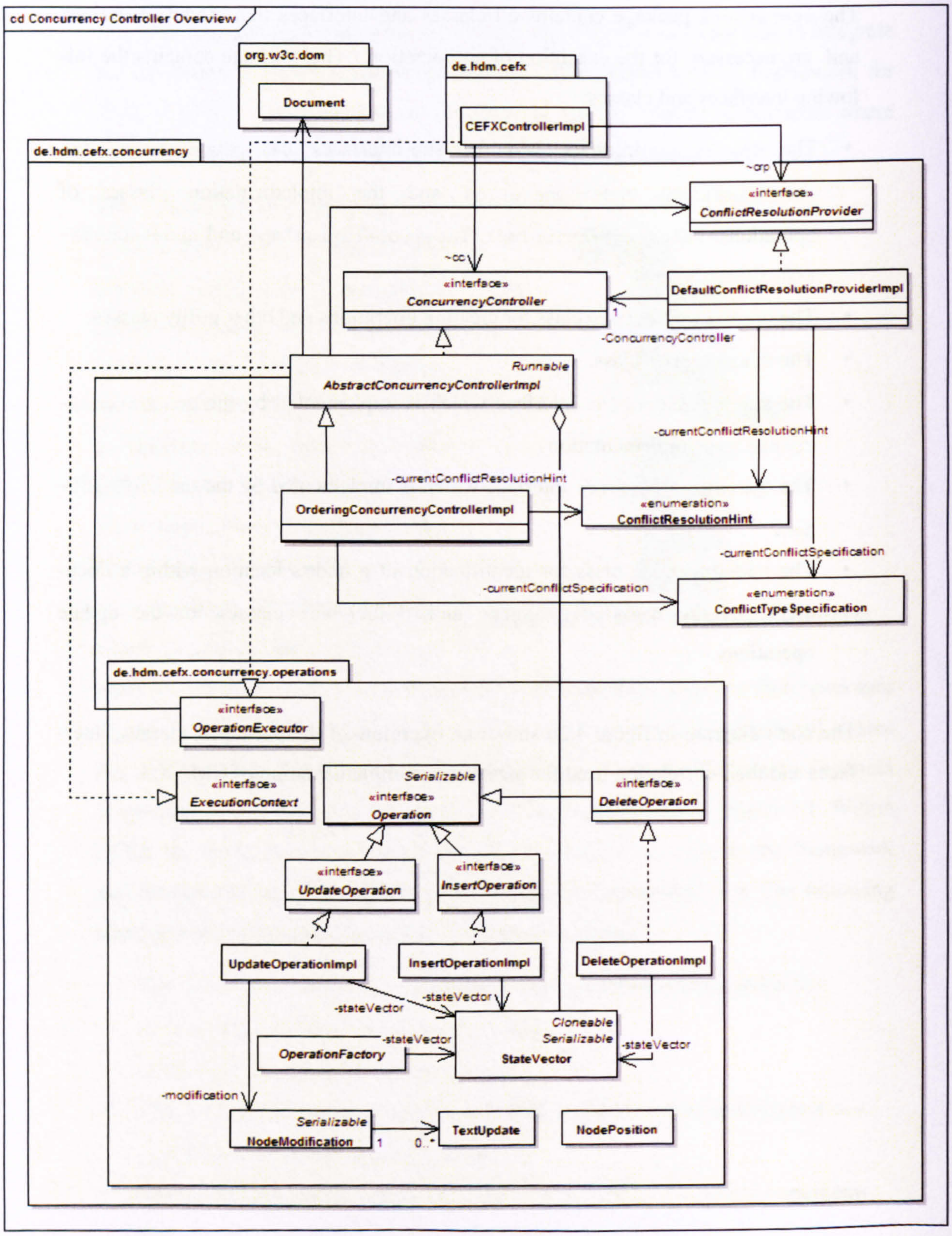

Figure 4.20: Overview of the CMAX classes and packages

4.4. Conclusions

The Java software components that are responsible for the consistency maintenance are the `ConcurrencyController` and the `ConflictResolutionProvider`. They were tested using simulation software that was specially developed for this purpose. After testing and refinement, the CMAX algorithms were successfully integrated into the Collaborative Editing Framework for XML (CEFX) which is fully discussed in chapters 5 and 6.

Chapter 5. The Collaborative Editing Framework for XML

5.1. Motivation

Cooperative work is a day-to-day activity in many areas. Software development teams cooperatively develop applications or write documentation. Engineers cooperatively design a circuit diagram or work together on a 3D model of a new machine. The documents that are cooperatively edited range from simple text documents over 2D graphics to complex 3D models.

However, software support for real-time group editing in commercial applications today is uncommon. Thus support for collaboration is often limited to turn-taking, split-combine and copy-merge. Existing real-time group editors - derived from research projects - are usually very specialized and despite latest achievements in the research field of Computer Supported Cooperative Work, many such systems suffer from a lack of user acceptance in the professional area. One reason for this seems to be a low motivation of users to learn new user interfaces and application functions if they cannot see their personal benefit in the collaboration features (Grudin 1994). Another reason may be that existing real-time group editors generally cannot compete with established single-user editors regarding the functionality and usability (Xia, Sun et al. 2004).

A more promising approach therefore is to extend accepted single-user editing applications with collaborative real-time editing functionality. The difficulty with this approach is to extend an application transparently (that is without modifying the application's source code) with as little effort as possible and at the same time providing the best support for real-time collaboration.

The idea of the Collaborative Editing Framework for XML (CEFX) is to provide a simple and flexible foundation for developing new collaborative real-time editing applications or enhancing existing single-user applications with collaborative real-time editing functionality.

CEFX is composed of a number of loosely-coupled components. One component of this framework contains the CMAX concurrency control algorithm as described in

chapters 3 and 4. Another component provides application developers with a simple-to-use interface for integrating collaboration functionality into existing single-user applications. Thereby, a novel integration mechanism makes use of standards such as XML and the DOM API in order to support a variety of applications. Other components manage the transmission of operations over the network or provide methods for the integration of awareness mechanisms. The main component of CEFX is the `CEFXController` which manages the communication between all other components.

The framework is designed as a flexible and extendible system in order to support more applications, different work flows and requirements. The following section of this chapter discusses contemporary collaborative systems and frameworks that support extending single-user applications with collaborative functionality. Next, the framework architecture of CEFX and the main framework components are discussed.

5.2. Contemporary Collaborative Systems

Collaborative systems that allow the enhancement of a single-user application with collaborative functionality are typically classified into collaboration-transparent systems and collaboration-aware systems. Systems that provide methods to share a single-user application without changing the application are called collaboration-transparent. The applied methods are unknown to the application and its developers. Collaboration-aware systems integrate collaboration mechanism by changing the application so that the application is aware of these. This allows a tight integration of collaboration functionality into an existing application. The problem with the collaboration-aware approach is that it requires access to the source code of an application which in some cases may not be possible for "off the shelf" software products.

The collaboration-transparent approach does not require access to an applications source code. Two types of collaboration-transparent systems can be identified: application independent (generic) and application dependent systems. Application independent systems do not know the shared application. They work on basis of transmitting low-level input/output data such as key-strokes, mouse movements and display pixel data. Application dependent systems are used to enhance a specific application with collaboration functionality and have knowledge of the application specific data model.

Examples of contemporary application independent and collaboration-transparent systems are application sharing environments such as NetMeeting[45], VNC[46], HP Remote Graphics[47] or Netviewer[48]. They allow sharing the view of any single-user application among a group of users. Such systems use a centralized architecture and transmit mouse movements, key strokes and screen images via the network. All users see exactly the same view of the shared application at the same time (strict WYSIWIS) but only one user at a time can interact with the application. In order to coordinate the access to the application often a direct communication between the users (via phone or chat) is necessary. These systems are useful and effective for tightly coupled collaborative work, where independent interaction is not wished or not required. Multi-user free interaction where each user can individually, for example, work on a different part of a shared document is not supported.

An example for an application dependent system is the CoWord system by Xia, Sun et al. (2004). Xia, Sun et al. propose the Transparent Adaptation (TA) approach for the extension of single-user applications with collaboration functionality. In CoWord, they have extended the Microsoft Word application transparently by making use of the Microsoft application and execution environment APIs (Application Programmers Interface). The TA approach requires each application to be adapted before being shared. The user actions performed on the word document are thereby intercepted and translated by the Adaption Layer into operations for the Operational Transformation (OT) Layer, which is responsible for maintaining the consistency of the shared document. The adaptation of an application requires the developer to have a detailed knowledge of the application and execution environment specific API. Additionally an interpretation of the user actions in relation to the current application contexts is required. Operational Transformation (OT) (Sun, Jia et al. 1998) is used for the concurrency control of a shared Word document. This requires to map each operation executed on the application's data model into an operation that can be processed by the OT concurrency control mechanism. This is the responsibility of the

---

[45] Microsoft NetMeeting. http://www.microsoft.com/windows/netmeeting/, retrieved October 30, 2007

[46] Real-VNC. http://www.realvnc.com/, retrieved October 30, 2007

[47] HP Remote Graphics. http://h20331.www2.hp.com/Hpsub/cache/286504-0-0-225-121.html, retrieved October 30, 2007

[48] Netviewer. http://www.netviewer.de/, retrieved October 30, 2007

Collaboration Adapter in CoWord.

However, mapping user actions to OT operations can become complex and requires that the application's data model supports positional addressing of objects, which may not be feasible for complex 3D modelling applications. These limitations of CoWord are not inherent to the OT approach, but come from the design choices made concerning the integration of concurrency control into an application.

The collaboration system underlying CoWord is collaboration-transparent, supports relaxed WYSIWIS and can also be used in a collaboration-aware system design. For the collaboration-transparent approach CoWord requires the execution environment and the single-user application to provide a suitable API which can be used to intercept and replay user input events and whose data and operational models are adaptable to that of the underlying OT technique. If an application and the execution environment provide a suitable API, the greatest effort lies in implementing the translation of the user actions into operations required by the OT Layer.

Other collaboration-transparent systems such as the one presented by He, Han et al. (2004) use a similar technique as in CoWord to enhance a single-user application with collaborative functionality. Low-level I/O events such as keyboard and mouse events are intercepted and translated into semantic commands. A so called Communicator collects high-level messages (such as CAD commands and model data) and low-level messages and transmits those over the network to the other collaborating sites. There they are translated into execution environment GUI commands or application specific API calls.

A similar approach is proposed by Li, Li et al. (2003) called Intelligent Collaboration Transparency (ICT). The focus of their work is on sharing heterogeneous applications of the same application family. They propose a system that allows extending single-user applications such as GVim and MS Word. For each application a so called ICT agent is implemented that captures events from the operating system and the application, translates them into semantic operations and then transmits those to the other collaborating sites, where the events are replayed in the form of a sequence of editing events. Their event capture and replay mechanism makes intensive use of application and operating system specific APIs and thus suffers from the same problems as the TA approach in terms of implementation complexity. Additionally the

complexity is increased by supporting heterogeneous applications. This requires a formalisation of application semantics in order to be able to translate the user actions of one application to the related user actions of another application.

The high implementation effort was one of the reasons for the second generation of the ICT project, ICT2. In contrast to their previous work, ICT2 does not attempt to intercept and understand the operating system level events. Instead it uses an adapted version of the "diffing" algorithm (Myers 1986) to derive the editing sequences between document states (Lu, Li et al. 2004). However, this new approach is not suited for fine-grained real-time group editing such as TA and ICT, because of its limitations in terms of performance. The support for heterogeneous applications is limited to those that have the same coding system. Sharing a document between, for example, Latex and Word is not supported. The diff algorithm that is applied supports text documents only. In order to support structured and formatted documents, more sophisticated diffing algorithms would be required (Li and Lu 2006).

The Flexible JAMM (Java Applets Made Multi-User) project uses a different approach (Begole 1999). Single-user applications are enhanced by replacing selected single-user components of the shared application with multi-user versions. This approach requires the underlying execution environment to meet certain conditions such as capabilities for process migration, run-time component replacement, dynamic binding and user input events interception and replay. The Java run-time environment is platform independent and meets these conditions. The common interface of JAMM applications is Java Swing and Java Object Serialization (JOS). In contrast to the other mentioned systems, the JAMM system does not require the development of a translation layer in order to convert user actions into application semantic commands or API calls, as long as an application is based on Swing and all application classes are serializable. Although the number of Java Swing based applications has increased in recent years, the number of single-user applications that fulfil the mentioned requirements is small.

To summarise, all approaches use a specific API in order to extend a single-user application. TA and ICT both use an API on the operating system as well as at the application level. The same is true for the approach of He, Han et al. (2004). JAMM uses an API on the level of the runtime's graphical user interface (GUI) library. The

first three approaches face the problem of implementation complexity for each new application that is to be extended. The JAMM approach has the problem of being dependent on the fulfilment of certain runtime requirements.

The goal of the approach in this thesis is to reduce the complexity of integrating a collaboration framework into a single-user application and provide a solution that is more general, supporting many different types of applications. We argue that this can be achieved by using aspect-oriented programming (AOP) and concentrating on the application's data model instead of an application, operating system or GUI library API.

The application data model describes how data is represented and used. For example in a text editing application, the data model represents the text that is edited. The structure of the data model can thereby be different to its visual representation. One aspect of an application is the manipulation of the data model. The code for updating or querying the data model can be distributed within the entire application. In the terminology of AOP such aspects are called cross-cutting concerns. In this approach firstly, these cross-cutting concerns or system-level-concerns within an application are identified. Secondly, so called 'advices' that create events for the underlying collaboration framework in order to synchronise the data model between the different sites are defined. This has the advantage that once developed, advices can be reused for all applications that use the same methods for the manipulation of their data model. This is for example the case for all applications using the Document Object Model (DOM) as a standard interface for the manipulation of XML content. Single-user applications that use the DOM as their internal data model can be easily extended without the effort of implementing a translation layer. Another advantage is that heterogeneous applications can be used to collaboratively work on a shared document because they only share a common interface, the DOM. Although new advices have to be developed for applications that use other methods for the data model manipulation, it is assumed that the implementation effort is lower in comparison to other methods.

The number of existing and emerging XML applications and thus the number of single-user applications using XML DOM as data model (native XML applications) is growing. Today, the majority of these applications are general XML editors or spe-

cialized SVG graphic editors (such as Sketsa[49], Amaya[50] and GLIPS[51]). Another example for an application that extensively uses the DOM is OpenOffice. OpenOffice uses OpenDocument[52] - an XML document format - as native file format.

An increasing number of applications exist that do not necessarily use XML DOM as their internal data model, but provide a DOM API that allows directly accessing and manipulating the internal data. These applications can also easily be extended by CEFX without requiring the difficult implementation of a translation layer.

In the case where an application is not based on XML and does not provide a DOM API, CEFX can also be used. This can be achieved by using similar methods as in the Transparent Adaptation approach of CoWord or other transparent approaches. In this case it would be necessary to implement a translation layer for converting user actions into XML operations for the CEFX collaboration layer. Technologies such as XML binding can dramatically reduce the time and effort of translating an internal data structure to the XML data model.

The following sections of this chapter discuss the framework architecture and its components. Chapter 6 discusses the implementation issues when developing CEFX. The aspect-oriented approach to integrating CEFX into an existing single-user application is discussed in chapter 7.

5.3. CEFX Software Architecture

The Collaborative Editing Framework for XML is based on a logical hybrid-architecture. The hybrid architecture is a mixture of both the centralised and the replicated architectures (see chapter 1.6.3). Each site holds a client and a server process as well as a copy of the shared data resource (the XML Document). Additionally a server site exists holding both the shared data resource and a server process. All operations are executed locally before they are sent to the other sites to be executed, just like in

---

[49] Sketsa. SVG Graphic Editor. http://www.kiyut.com/products/sketsa/index.html, retrieved October 30, 2007

[50] Amaya. W3C's Editor/Browser. Open Source. http://www.w3.org/Amaya/, retrieved October 30, 2007

[51] GLIPS Graffiti Editor. Open source SVG graphics editor. http://glipssvgeditor.sourceforge.net/, retrieved October 30, 2007

[52] OASIS Open Document Format for Office Applications (OpenDocument). OASIS Standard May 2005

the replicated architecture approach. Additionally the operations are also sent to the server site. There they are executed as well. Turning to the question of the responsiveness, as in replicated systems, the hybrid approach has the advantage of having a central site that holds the current correct version of the document. If a new site joins the session, a copy of this version can be obtained easily.

In a logical hybrid-architecture, a central server does not necessarily exist physically. It can exist for example only as a logical server running on any client of the collaboration. This means that any client can be configured to additionally play the role of the centralized server. The advantage of this is that no extra server hardware has to be provided, because in some environments it may be difficult to do so for security, monetary or political reasons. Figure 5.1 shows a deployment diagram with the server and two client sites.

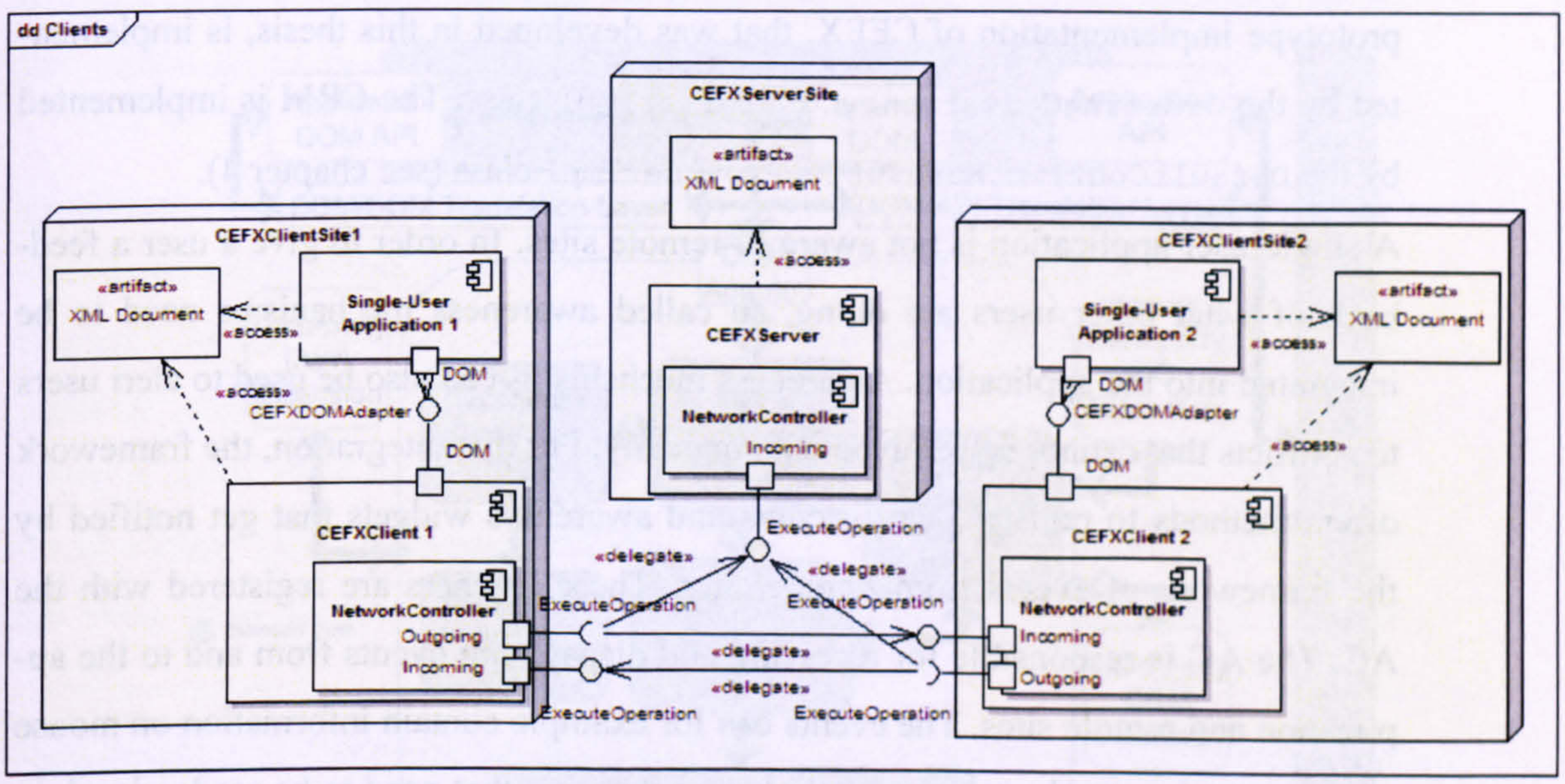

Figure 5.1: Hybrid architecture with server and clients

Each of the `CEFXClient` components (`CEFXClient 1` and `CEFXClient 2`) and the `CEFXServer` component have a `NetworkController` component. The `NetworkController` of each client owns an incoming and outgoing port. The incoming port represents the server process of a client providing an interface for other clients to connect to and accepting incoming connections in order to receive remote operations. The outgoing port represents the client process. It is used to connect to other clients and to send operations to them. The `CEFXServer` component's `NetworkController` only has an incoming port providing an interface that allows other client to connect

and send operations to the server. Additionally to the interfaces shown in figure 5.1, the `CEFXClient` and the `CEFXServer` components have interfaces to join and leave a session, to retrieve and send XML documents and to propagate and receive awareness events. These interfaces are discussed in detail in chapter 6.

5.3.1. CEFX components

The main CEFX components are the concurrency controller (CC), the conflict resolution module (CRM), the awareness controller (AC), the awareness widgets (AW), the DOM adapter (DA), the CEFX controller (CEFXC) and the network controller (NC). The CC of the framework is responsible for maintaining the consistency of the document and uses the NC to transmit and receive editing events to and from remote sites. The CRM defines the rules that are applied if a conflict occurs. The CC in the prototype implementation of CEFX, that was developed in this thesis, is implemented by the `OrderingConcurrencyControllerImpl` class. The CRM is implemented by the `DefaultConflictResolutionProviderImpl` class (see chapter 4).

A single-user application is not aware of remote sites. In order to give a user a feedback of what other users are doing, so called awareness mechanisms need to be integrated into the application. Awareness mechanisms can also be used to alert users to conflicts that cannot be resolved automatically. For this integration, the framework offers methods to register own listeners and awareness widgets that get notified by the framework of events from remote sites. These listeners are registered with the AC. The AC is responsible for receiving and dispatching events from and to the application and remote sites. The events can for example contain information on mouse movements, key stroke events or other control events that need to be made visual, in some way or other, to the user.

The DA connects an application with CEFX. It is responsible for creating operations on the basis of application events, for example user actions, and delegates the execution of the operations to the CEFX controller. Three different ways of integrating the DA into an application exist. The DA can, for example, be directly connected to the single-user application's internal data model (the DOM). This is achieved, in the prototype implementation of this thesis, by using the aspect oriented programming approach. In other cases where the application provides a DOM API, the DA can be

connected with the application through a DOM/DOM translation layer that forwards the application's data model events to the DA and vice versa. Applications that do not use the DOM internally for the manipulation of their data and do not provide a DOM API can also be extended with CEFX. In that case the approach to connect the DA with the application is similar to the transparent adaptation approach used in CoWord. This requires the implementation of a DOM/API translation layer which translates application and runtime (OS) events into DOM events and vice versa.

The CEFXC is responsible for managing the client's session and delegates editing and awareness events to the corresponding modules of the framework such as the CC, the AC or the NC.

Figure 5.2 shows an overview of the main CEFX framework components and their interrelation.

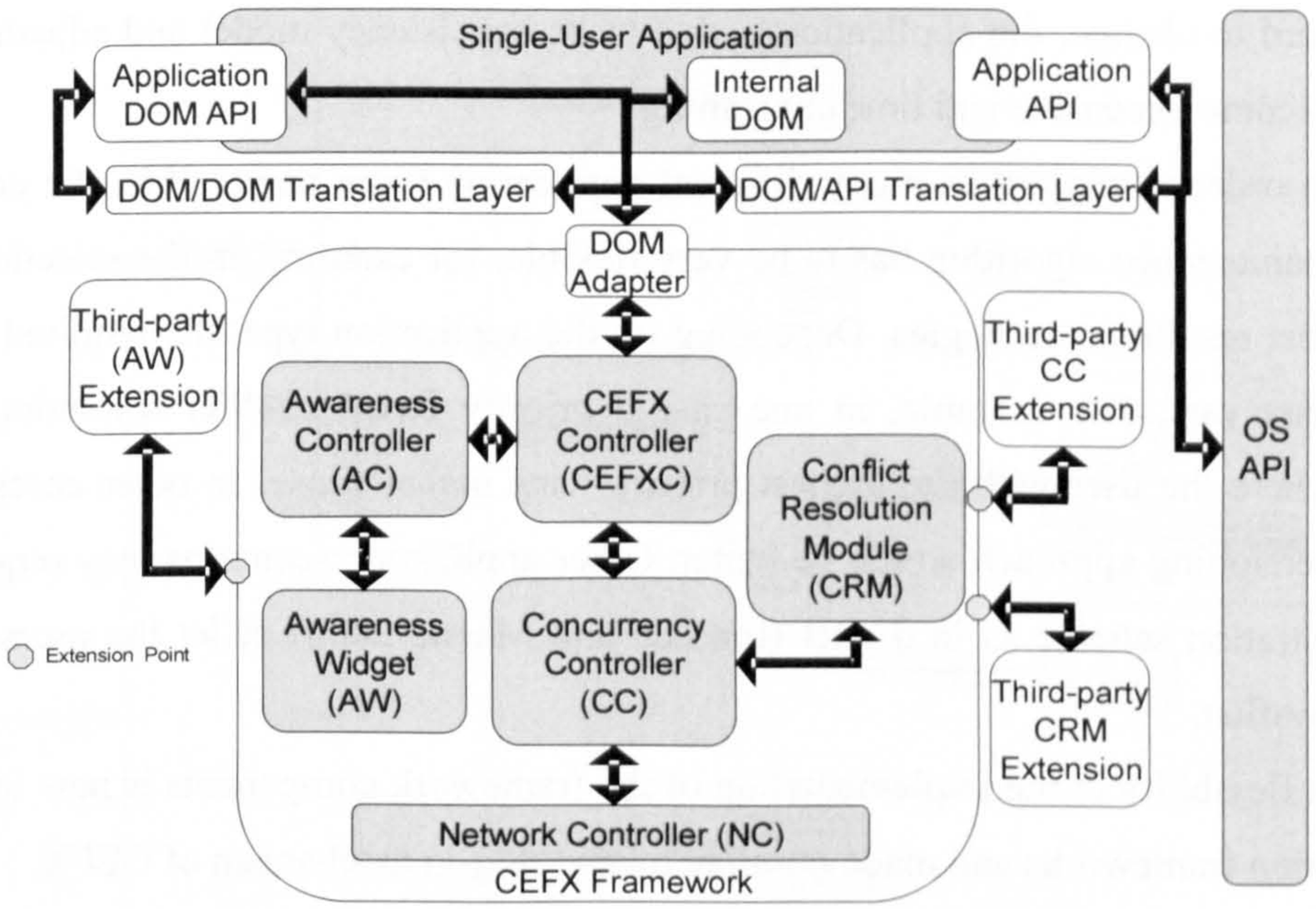

Figure 5.2: Main CEFX components

An application can use the default framework components such as awareness widgets or provide its own and register them with the framework. This is possible due to the plug-in architecture of CEFX. Each of the main components shown in figure 5.2 are plug-ins that can be configured using the so called extension points mechanism. The framework offers extension points to extend or replace awareness widgets, the con-

currency controller and conflict resolution module implementations or the network controller.

The following section discusses the plug-in mechanism used in CEFX for the flexible extension of the framework. Next the tasks of the DOM adapter, the awareness controller and widgets, the network controller and the CEFX controller components are further discussed.

5.3.1.1. The Plug-in Mechanism

Enhancing single-user applications with collaborative functionality or developing new real-time collaborative applications is difficult and time consuming. Even if an application is developed using a collaborative editing framework a lot of coding is necessary. If the framework does not fulfil all requirements of the application, adjustments and changes to the source code are indispensable. With a framework that is hard to change, the application is tied to its consistency model and adjustments can become expensive and time consuming.

In order to support as many different application types as possible, the consistency maintenance algorithm has to be very flexible, for example, in the selection of conflict resolution strategies. Depending on the application type, the required strategies may vary. For example, in one case a priority based conflict resolution strategy, where the user with the highest priority wins makes sense. In other cases, a multi-versioning approach would be better. Other application scenarios may require an arbitration scheme as in dARB (Ionescu and Marsic 2000) or let the users solve the conflict.

A flexibility in the implementation of the framework components is new to collaboration frameworks and made possible by the plug-in mechanism of CEFX.

Plug-ins are software extensions that are implemented in compliance to a certain software interface (a contract defining how something is to be implemented). This allows a plug-in to be used by another software without knowing any implementation details of the plug-in. The other software, in this case CEFX, only knows the interface of the used plug-in. So called extension points are used to inform CEFX about which plug-ins should be used. An extension point is a contract which consists of an extension point declaration (defined in an XML file) and a Java interface definition (Java programming language source code).

CEFX defines extension points for the following framework components:

- Concurrency controller
- Awareness controller
- Awareness widgets
- Conflict resolution module
- Network controller

The extension points are declared in the CEFX configuration XML file (cefx.xml) which is structured as defined by the CEFX extensions XML Schema (cefxextensions.xsd). Figure 5.3 depicts the extensions XML Schema definition.

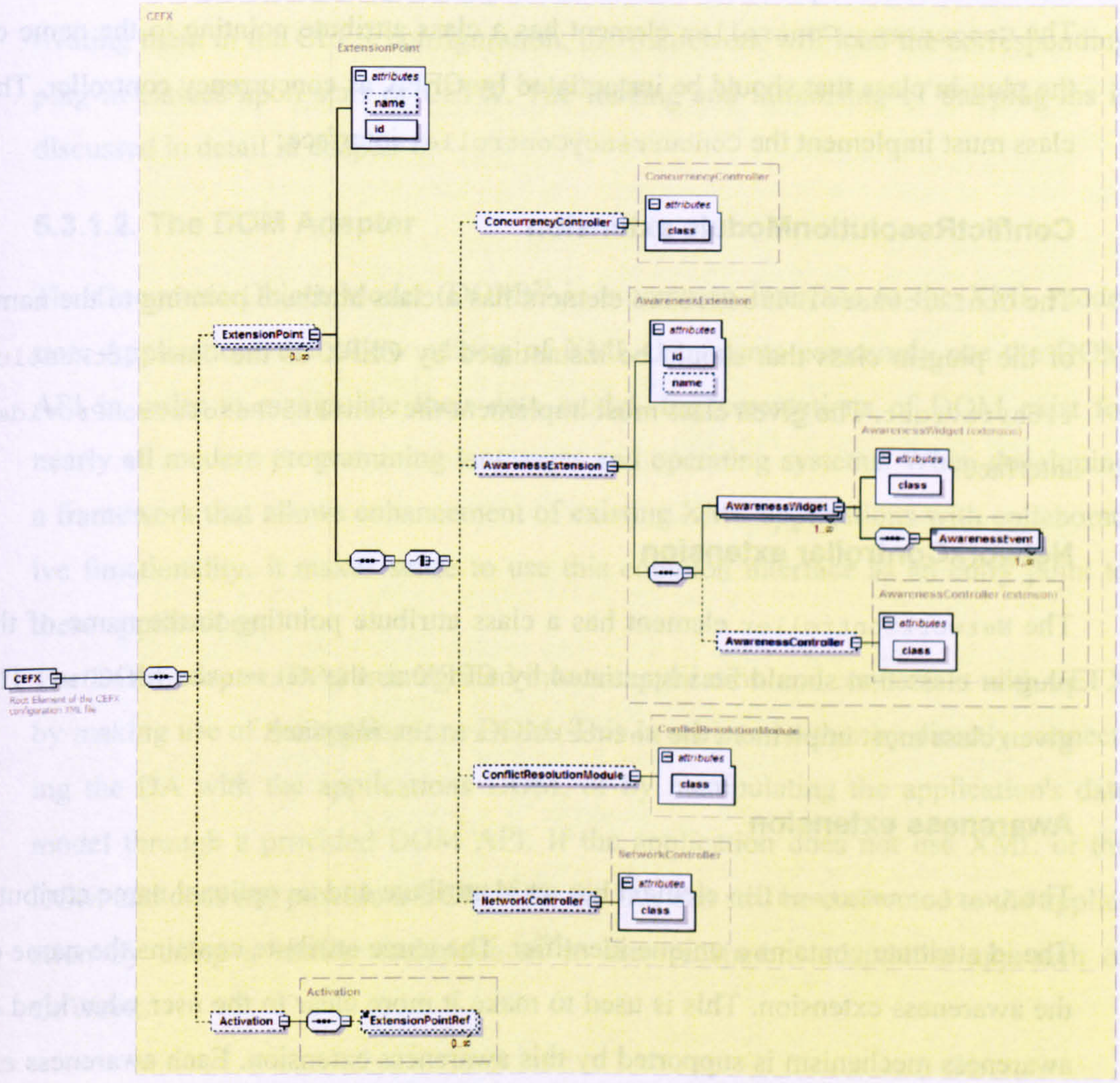

Figure 5.3: CEFX extensions XML Schema

The extension points declaration for CEFX is separated into two parts. The first part

is the extension point declaration itself (complex type `ExtensionPoint`), the second part is the activation declaration (complex type `Activation`). The extension point declaration part of the CEFX configuration defines which plug-ins exist. In order to make CEFX use a certain plug-in extension, the extension has to be activated in the activation declaration part of the CEFX configuration. This allows it to declare a set of extension plug-ins and switch between configurations by changing the set of active plug-ins. Each extension point has an id attribute that allows to identify the activated extensions.

ConcurrencyController extension

The `ConcurrencyController` element has a class attribute pointing to the name of the plug-in class that should be instantiated by CEFX as concurrency controller. The class must implement the `ConcurrencyController` interface.

ConflictResolutionModule extension

The `ConflictResolutionModule` element has a class attribute pointing to the name of the plug-in class that should be instantiated by CEFX as the `ConflictResolutionProvider`. The given class must implement the `ConflictResolutionProvider` interface.

NetworkController extension

The `NetworkController` element has a class attribute pointing to the name of the plug-in class that should be instantiated by CEFX as the `NetworkController`. The given class must implement the `NetworkController` interface.

Awareness extension

The `AwarenessExtension` element has an id attribute and an optional name attribute. The id attribute contains a unique identifier. The name attribute contains the name of the awareness extension. This is used to make it more clear to the user what kind of awareness mechanism is supported by this awareness extension. Each awareness extension has a corresponding awareness widget declaration and an optional awareness controller declaration. The `AwarenessWidget` element has a class attribute and one or more `AwarenessEvent` elements. The class attribute points to the name of the plug-in

class that should be instantiated by CEFX as `AwarenessWidget`. The class must implement the `AwarenessWidget` interface. The `AwarenessEvent` element contains a text that identifies the type of event that this awareness widget will listen to. Awareness events are, for example, mouse selection events, mouse movement events or key stroke events. Whenever one of the here defined events occurs, the corresponding awareness widget will be notified by CEFX. The `AwarenessController` element has a class attribute that points to the name of the plug-in class that should be instantiated by CEFX as the `AwarenessController`. The given class must implement the `AwarenessController` interface.

After defining the used plug-in extensions via the extension point declaration and activating them in the CEFX configuration, the framework will load the corresponding plug-in classes upon start of CEFX. The loading and initialising of the plug-ins is discussed in detail in chapter 6.

5.3.1.2. The DOM Adapter

The Document Object Model (DOM)[53] is a common interface to the XML model tree. Applications that allow editing of XML documents commonly use the DOM API in order to manipulate their data model. Implementations of DOM exist for nearly all modern programming languages and operating systems. When developing a framework that allows enhancement of existing XML applications with collaborative functionality, it makes sense to use this common interface as an entry point to these applications.

The DOM adapter (DA) is integrated into an application and connects it with CEFX by making use of the applications DOM. This is achieved either by directly connecting the DA with the applications DOM, or by manipulating the application's data model through a provided DOM API. If the application does not use XML or the DOM and does not provide a DOM API, the DA can still be connected to the application by using a similar technique as in the transparent adaptation approach of CoWord.

---

[53] W3C Document Object Model. Platform- and language-independent standard object model. http://www.w3.org/DOM/ , retrieved October 30, 2007

A shared data model

In this work's prototype implementation of a collaborative SVG editing application, the application is extended by directly connecting the DA to the applications DOM (further discussed in chapter 7). In that case the single-user application and the DA share the same data model (represented by a `org.w3c.dom.Document` object). The `org.w3c.dom.Document` interface defined by the W3C DOM API specification provides methods for accessing and manipulating the XML document's content. When the XML document is loaded and the Document object is created, both the application and the DA retrieve a reference to it. The DA then passes the reference to the `ConcurrencyController` component of CEFX.

Whenever the document is manipulated by the application as a result of a user action, the DA is notified and creates a corresponding operation. The operation is then handed over to the `CEFXController` which then calls the corresponding `executeOperation(...)` method of the `ConcurrencyController` (see chapter 4).

When a remote operation arrives at a site, it is executed on the shared `Document` object directly by the `ConcurrencyController`. The DA thereafter notifies the application of a change of the data model so the application can repaint its view and present the changes to the user.

DOM/DOM translation

Some applications may not use the DOM internally but provide a DOM API as an interface for other application to their data model. This is, for example, the case for a number of web browsers such as Microsoft Internet Explorer and FireFox. The provided DOM API usually allows to register DOM listeners that are notified if the document's data is changed, for example by a user action. They also allow other applications to access their data model through the provided DOM API and manipulate it. In order to connect such an application with CEFX, a DOM/DOM translation layer (see figure 5.2) needs to be implemented, which registers itself as listener with the applications DOM and forwards DOM events to the DA. The DA then shares a `org.w3c.dom.Document` object with that translation layer and any changes made to the applications document are also executed on the DA and the translation layers document. When remote operations are executed on the shared document, the translation

layer translates those manipulations into calls to the application's DOM API and reproduces the changes there. In this way the documents are synchronised by the DOM/DOM translation layer without a direct connection of CEFX and the application.

DOM/API translation

As mentioned before, it is also possible to use CEFX with applications that do not use the DOM internally and do not provide a DOM API. In this case the application that is to be extended must provide another API that allows manipulation of the application's data model. Microsoft Word for example provides such an API. In order to connect such an application with CEFX, a DOM/API translation layer (see figure 5.2) would have to be implemented that - similarly to the above approach - shares a data model with the DA but this time intercepts application and operating system events such as mouse movements and key strokes and translates them to DOM manipulation actions on the shared document. It may be very elaborate to do so, but it is feasible as similar approaches such as the CoWord approach show. The difference to CoWord here is that the events are not translated into operations for the operational transformation layer but into operations that manipulate an XML document. The XML document that is used in such a case should either model the extended applications data model as close as possible or be a document that allows the translation layer to easily translate the remote operations into calls to the application's API.

5.3.1.3. The CEFX Controller

The `CEFXController` takes care of the session handling and the server connection. When a document is opened, the `CEFXController` checks, if the client is already connected to a session with the server. If no session exists, the controller connects the client to the server and opens a new session.

A document that is opened for the first time in a collaborative editing session, is opened locally by the `CEFXController` and then send to the server. The server then stores this document. The next time the document is opened, the `CEFXController` loads it from the server instead of the local file system.

The `CEFXController` is created in the initialisation phase of the DA and loads all plug-ins - as configured in the CEFX configuration file - and initialises them. This is

done with the help of the ExtensionRegistry which is further discussed in chapter 6. The controller has a connection to every component of the system. It is responsible for delegating the execution of operations to the concurrency controller and allows retrieval of a reference to each of the other framework components. The CEFXController implements the OperationExecutor interface (see chapter 4) and thus acts, from the perspective of the concurrency controller, as a client.

5.3.1.4. The Network Controller

The main task of the NetworkController is to connect the client to the server and to the other clients and propagate and receive operations. When it receives a remote operation it delegates it to the CEFXController which then further processes the operation. The NetworkController provides certain network protocol interfaces that allow other clients to connect to it. The prototype implementation of the CEFX NetworkController uses the Java Remote Method Protocol (JRMP). This allows connecting the clients (and the server) via a TCP/IP based network. JRMP is a protocol often used in professional applications.

5.3.1.5. The Awareness Controller

Awareness mechanisms are an important part of a collaborative application. They provide a user with information on other user's actions within a collaboration session. CEFX therefore supports the integration of awareness mechanisms into the framework. The AwarenessController component is responsible for delegating editing events such as mouse movements and key strokes, that a user might be interested in, to the corresponding awareness widgets. Awareness widgets are registered with the AwarenessController and provide certain methods that allow the AwarenessController to receive information on the kind of events each awareness widget is interested in. For example, an awareness widget that informs the user on mouse events of other users within the editing context is informed by the AwarenessController of such events. The AwarenessController receives those events from the other clients and is responsible for propagating such events to all other clients within a session. The prototype implementation of CEFX supports two types of events: mouse selection and operation execution. Whenever a user selects an object within the document, or executes an operation, the AwarenessController creates the cor-

responding awareness event and propagates it to the other clients. When another client receives such an event, the corresponding awareness widget is notified. The awareness widget then visualises the event.

5.4. Conclusions

The work in this thesis successfully employed, for the first time, the flexible plug-in concept in a collaborative editing framework whereby each component can be easily extended or replaced with a new implementation. This novel technique resulting from this research will potentially enable more efficient adaptation of the framework to specific application requirements.

A new method of using the Document Object Model (DOM) as a standard interface for the integration of collaboration functionality into an application was developed and successfully implemented as described in chapter 6.

This simplified the coding and potentially reduces the development time compared with other approaches. CEFX will provide a simple-to-use application programmer's interface (API) that enables the development of new collaborative XML editing applications or the extension of existing single-user applications as discussed in chapter 7.

Chapter 6. Implementation of the CEFX system

The CEFX implementation is split into two parts: the implementation of the client part (client) and the server part (server). The client is used when developing a new collaborative real-time editing application or when extending an existing single-user application. The server is an application of its own and runs independent of the client. It can either be started on a separate computer or run in parallel to a client on the same computer.

In order to connect the clients and the server with each other, the Java Remote Method Invocation (RMI) API is used. Java RMI, is a Java application programming interface for performing the object equivalent of remote procedure calls. RMI will not be further discussed in this thesis, as it is a common method for implementing remote procedure calls[54].

The client and the server part of the CEFX software were developed using the Eclipse[55] Integrated Development Environment (IDE) and the Java programming language.

The main objectives when designing the Collaborative Editing Framework for XML (CEFX) were to:

- provide an easy to use application programmer's interface (API) allowing an application programmer to build new collaborative real-time editing applications without having to care about synchronisation and collaboration issues.
- provide an easy to use integration mechanisms allowing the easy extension of existing single-user editing applications.
- provide a flexible software architecture allowing the easy extension of CEFX with new functionality or concurrency control mechanisms.
- develop a platform independent software that can be used on various operating systems.
- achieve a good overall system performance that allows the user to work with a

---

[54] For more information on RMI please refer to http://en.wikipedia.org/wiki/Java_RMI, retrieved October 30, 2007

[55] For more information on Eclipse, see http://www.eclipse.org, retrieved October 30, 2007

CEFX based application as if it were a single-user application.

The following chapters discuss the software structure of CEFX, that is the implementation of the server and the client part of the framework, the different source packages and their contained components.

6.1. CEFX Client

The client part of CEFX is structured into several Java packages. The base package is the `de.hdm.cefx` package containing the `CEFXController` interface and its implementing class, the `CEFXControllerImpl` class. Below this package, the following packages exist:

- `concurrency`
- `concurreny.operations`
- `dom.adapter`
- `extension`
- `registry`
- `awareness`
- `awareness.event`
- `client`
- `client.net`
- `util`

In the following sections each of these packages and their contained classes are discussed. The `concurrency` and `concurrency.operations` packages contain all classes relevant to the concurrency control mechanism which were discussed in chapter 4.

6.1.1. The CEFX client base package

The `de.hdm.cefx` package contains the classes as shown in figure 6.1.

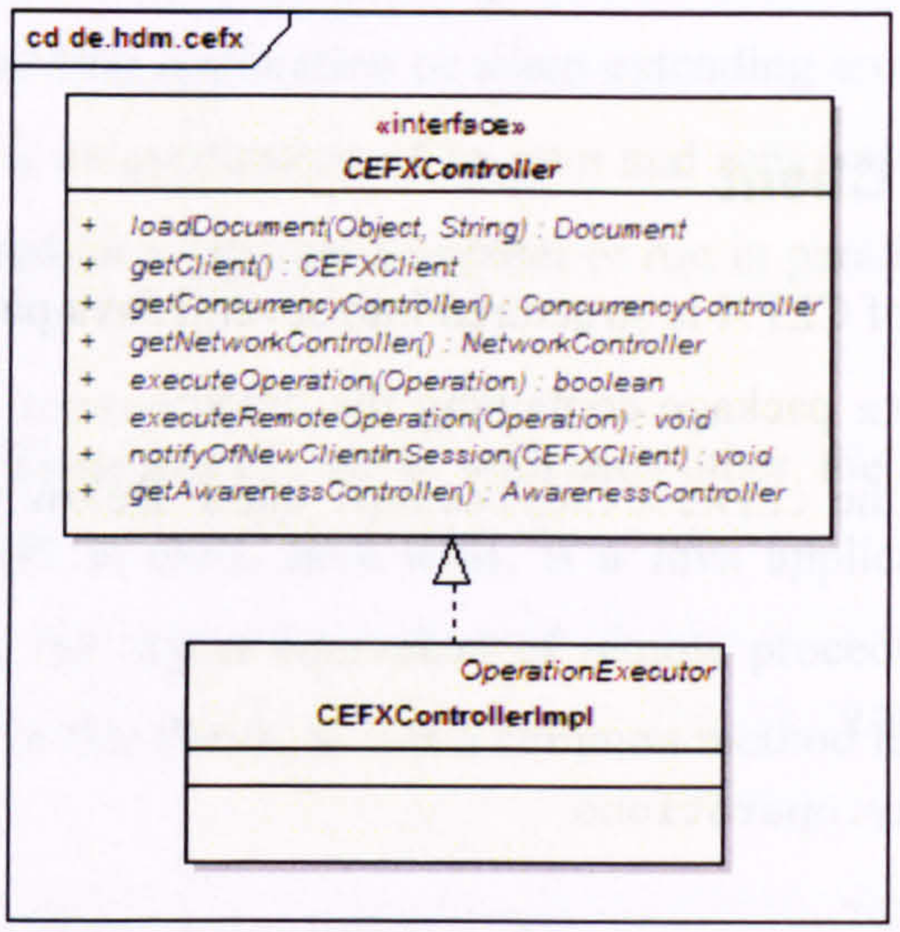

Figure 6.1: Classes in Java package de.hdm.cefx

The `CEFXController` interface is implemented by the `CEFXControllerImpl` class and defines the following methods:

- `boolean executeOperation(Operation operation);`
- `void executeRemoteOperation(Operation operation);`
- `CEFXClient getClient();`
- `ConcurrencyController getConcurrencyController();`
- `NetworkController getNetworkController();`
- `AwarenessController getAwarenessController();`
- `void notifyOfNewClientInSession(CEFXClient client);`
- `Document loadDocument(Object documentFactory, String documentURI);`

The method `executeOperation(Operation operation)` is called by the `CEFXDOMAdapter` after the user issued a change of the document in the application. The `CEFXDOMAdapter` creates an `Operation` object and passes it on to the `CEFXControl-`

ler which in turn delegates the execution of the local operation to the Concurrency-Controller.

Remote operations are received by the NetworkController and passed to the CEFX-Controller via the executeRemoteOperation(...) method. The CEFXController delegates the execution of the remote operation to the ConcurrencyController and notifies the CEFXDOMAdapter to refresh the application's user interface in order to visualise the changes to the user.

The CEFXController owns a CEFXClient object and the method getClient() is used by the NetworkController to retrieve this object. The CEFXClient object contains information necessary for connecting to the server and opening incoming connection ports (see chapter 6.1.1.3 and 6.1.1.4 for details).

The CEFXController initialises – next to the CEFXClient object - the ConcurrencyController including the ConflictResolutionProvider, the NetworkController and the AwarenessController. References to these components can be retrieved by using the methods getConcurrencyController(), getNetworkController() and get AwarenessController().

The method notifyOfNewClientInSession(CEFXClient client) is called by the NetworkController when a new client joins an open session. The CEFXController then adds the new client ID to the ConcurrencyController's state vector so that the new client is taken into consideration in the concurrency control. After this the state vector and the history buffer are cleared in order have the same default state at all editing sites.

It may be noticed that during the time when a new client joins an existing editing session and retrieves the current document from the server, no editing should take place. This is to make sure that each client and the server have the same initial document state when starting the concurrent editing session again. In order to prevent users from editing while a new client joins, the users should be notified. Additionally the user actions should be either blocked for that period of time or the actions should be buffered, so that they can be executed automatically after the joining sequence is completed. In the CEFX proof of concept prototype implementation it was not necessary to include such mechanisms. This is an open issue for future CEFX implementations.

6.1.1.1. Loading a document and initialising an *editing* session

Each client has a local document repository where documents are stored that can be edited collaboratively. The repository in this case is a folder called `CEFXRepository` on the local hard drive. If a document is contained in this repository and is opened by the user for editing purposes, the method `loadDocument(...)` of the `CEFXController` is called by the `CEFXDOMAdapter`. The method's `documentFactory` argument (of the type `Object`) is a reference to the editing applications DOM factory object. This can be an object of any type that implements the `org.w3c.dom.DOMImplementation` interface or has a method `createDocument(String uri)` returning a `org.w3c.dom.Document` object. The argument `documentURI` (of the type `String`) is a path to the document in the local document repository.

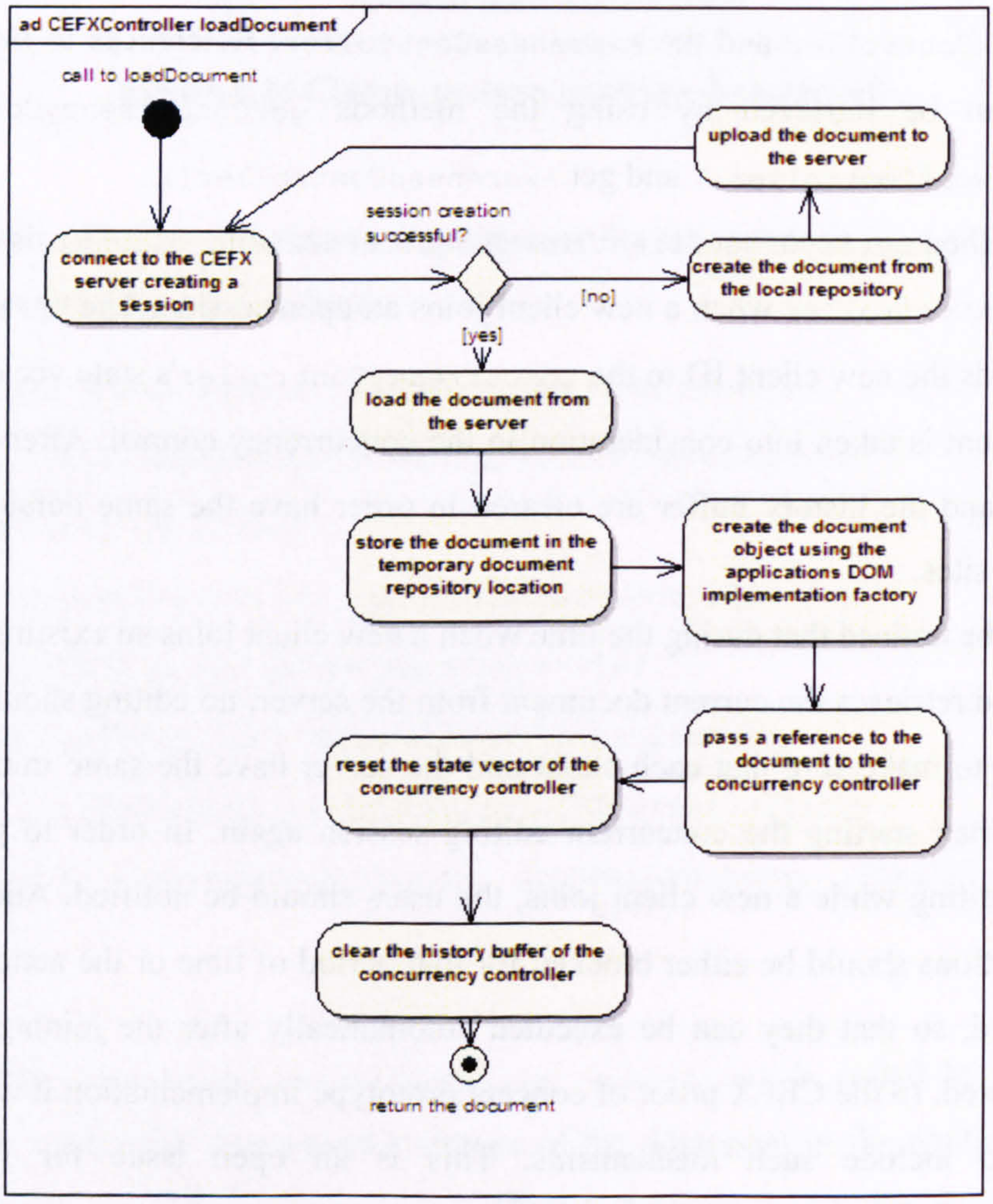

Figure 6.2: Execution step of the loadDocument(...) method

The steps shown in figure 6.2 are executed by the `CEFXController` when the `loadDocument(...)` method is called.

First the `CEFXController` connects to the server in order to create a new or to join an existing editing session. If the server already manages a session for the document, or if a new session can be created, the server returns a session object to the client and the `CEFXController` is notified that the connection has been successfully established. In this case the `CEFXController` will load the document from the server and then store it in a local temporary repository. If a connection could not be established and no session object was returned from the server this means that the server does not recognise the requested document and it has to be uploaded to the server in the first place.

The server requires a copy of the document in order to process it and add UUIDs to each element node. After uploading the document, the `CEFXController` tries to connect to the server again. This time connecting will succeed and the document can be loaded from the server. The next step is to create a `org.w3c.dom.Document` object by using the application's document factory object. Once the document object has been created, a reference to it is passed to the `ConcurrencyController` implementation of CEFX. Then the `ConcurrencyController` is prepared for collaborative editing by resetting the state vector and clearing the history buffer. The last step is to return the document object to the `CEFXDOMAdapter`. The `CEFXDOMAdapter` will then pass it to the application.

Figure 6.3 schematically depicts a successful session initialisation sequence. The calls (shown in the sequence diagram in figure 6.3) between the `NetworkController` and the `CEFXServer` are RMI network calls involving a number of other objects that are not shown here for simplicity reasons. It is worth mentioning that the document object from the server site is created by using the document factory provided by the standard Java 1.5 runtime environment (class `javax.xml.parsers.DocumentBuilderFactory`). At the client site, the application's document factory is used.

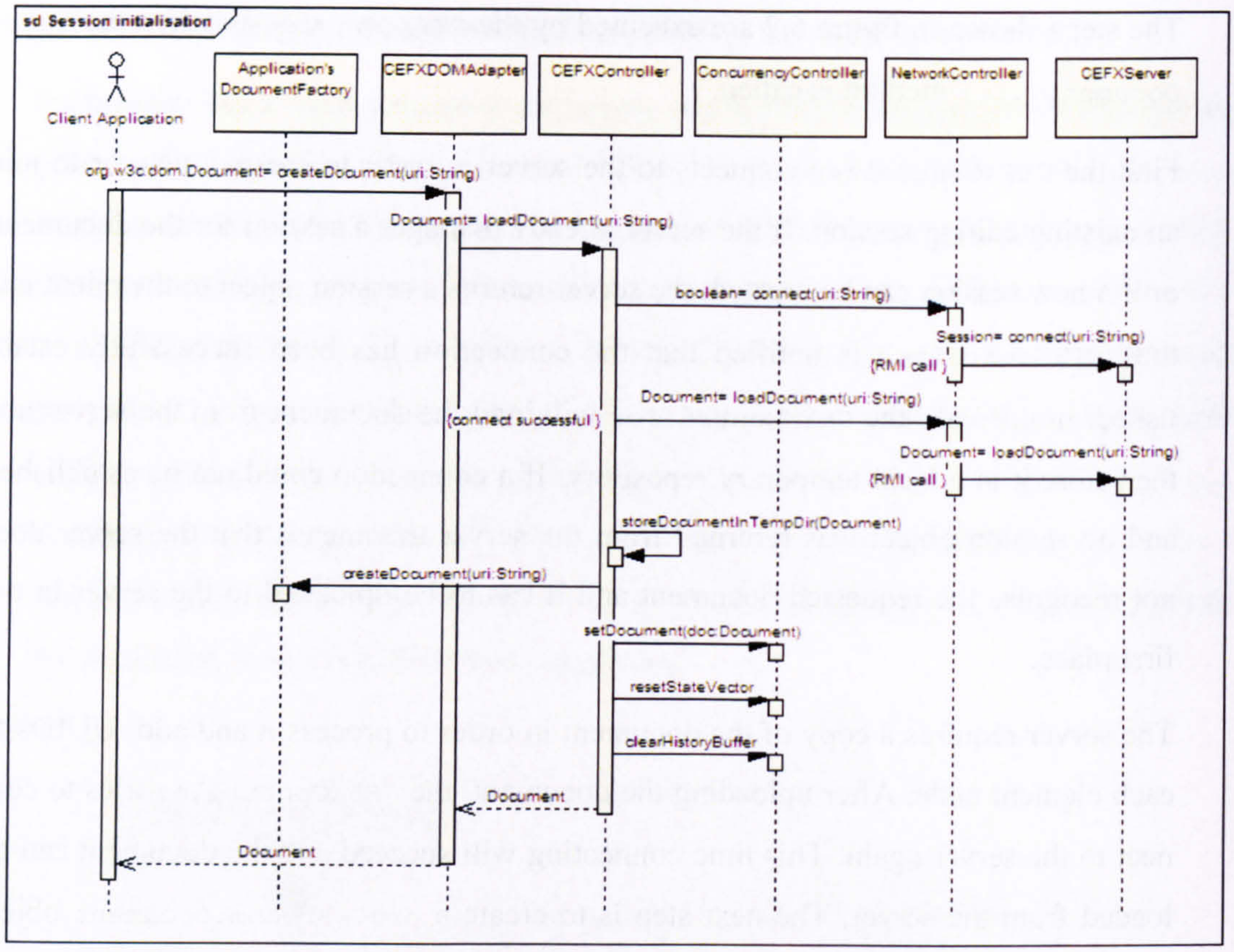

Figure 6.3: Sequence of calls when successfully initialising a editing session

Each application for a specific XML document type, for example SVG, DocBook or X3D might provide its own XML document object implementation. This is done to fulfil certain requirements that are necessary in order to, for example, render the XML document as graphic (as in the case of the Batik SVG library). However, the implemented document object - including its containing element nodes - must conform to the W3C DOM API. That is, the applications document object must implement the interfaces defined by the standard W3C DOM API. This allows it to be processed using the standard DOM methods. In order to integrate the client part of CEFX seamlessly into an editing application, it should be possible to share the document object with it in order to execute remote operations directly on the application's data model (in the case of a transparent adaptation).

As mentioned before, the server does not have a reference to the application's document factory. Thus the standard W3C DOM API is used to load and process an XML document at the server site, keeping the server independent of the XML editing application.

6.1.1.2. Initialisation of the CEFXController

When the `CEFXController` is initialised, it in turn initialises all other required framework components except for the `CEFXDOMAdapter` (which is responsible for initialising the `CEFXController`). First the `CEFXController` initialises the `ExtensionRegistry`. The `ExtensionRegistry` contains all information required for the initialisation of the other plug-in components such as the `AwarenessController` and `AwarenessWidget` components, the `NetworkController`, the `ConcurrencyController` and the `ConflictResolutionModule`. In order to initialise, for example, the `AwarenessController`, the `CEFXController` retrieves an `AwarenessExtension` configuration object from the `ExtensionRegistry`. From this `AwarenessExtension` configuration object the class name of the configured awareness class is retrieved. The next step is to create an object of this class by using the Java class loading functionality as shown in the below code example:

```
final String acClazz =
      acConfiguration.getAwarenessController().getClazz();
AwarenessController ac = null;
Class acClass = Class.forName(acClazz);
ac = (AwarenessController) acClass.newInstance();
```

The `acConfiguration` object in the above code example represents the mentioned `AwarenessExtension` object.

The same mechanism is used for the initialisation of each of the other framework components. Additionally the mentioned `CEFXClient` object is created. The `CEFXClient` object, the `ExtensionRegistry` and the configuration classes are discussed in detail later.

6.1.2. The dom.adapter package

The `dom.adapter` package contains the `CEFXDOMAdapter` interface and its implementing class the `CEFXDOMAdapterImpl` as shown in figure 6.4.

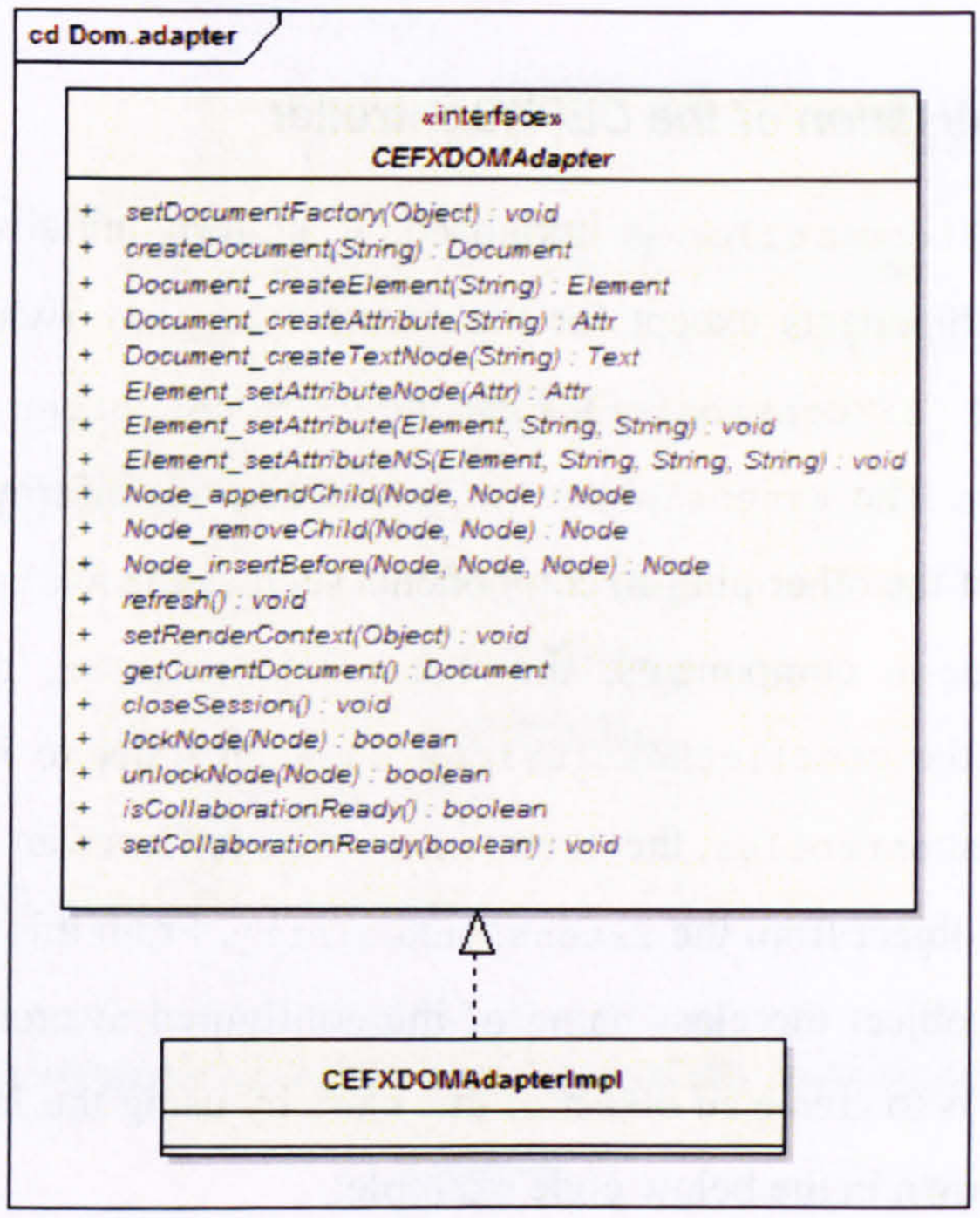

Figure 6.4: Package dom.adapter classes

The `CEFXDOMAdapter` interface defines a set of methods that correspond to the W3C DOM API methods for manipulating the content of an XML document. These methods are:

- `Element Document_createElement(String tagName);`
- `Attr Document_createAttribute(String name);`
- `Text Document createTextNode(String data);`
- `Attr Element_setAttributeNode(Attr newAttr);`
- `void Element_setAttribute(Element e, String attr, String value);`
- `void Element_setAttributeNS(Element element, String nameSpaceURI, String qualifiedName, String attributeValue);`
- `Node Node_appendChild(Node parent, Node newChild);`

- `Node Node_removeChild(Node parent, Node child);`
- `Node Node_insertBefore(Node parent, Node currentChild, Node oldChild);`
- `Document createDocument(String uri);`

These methods are used basically in the same way as the W3C DOM API is used when working with an XML document. This makes it easy for an application developer being familiar with the DOM API to learn how to use the `CEFXDOMAdapter` interface methods. The method `createDocument(...)` (as explained in chapter 6.1.1.1) is used to instantiate a `org.w3c.dom.Document` object. In order to modify the document the other listed methods can be used. For example the method `Node_appendChild(...)` is called to append a node to another existing node within the document. The `CEFXDOMAdapter` will in this case take care that the corresponding CEFX operation is created and passes it on to the `CEFXController`.

The other methods of the `CEFXDOMAdapter` interface are used, amongst other things, for initialising the framework and the editing session, checking the framework status and for the CEFX locking feature allowing to lock certain parts of a document.

- `void setDocumentFactory(Object factory);`
- `void setRenderContext(Object renderContext);`
- `void refresh();`
- `void setCollaborationReady(boolean readyState);`
- `void closeSession();`
- `boolean lockNode(Node node);`
- `boolean unlockNode(Node node);`

The method `setDocumentFactory(...)` is important for the framework initialisation and is used to pass a reference of the application's document factory to CEFX.

The render context (passed to the `setRenderContext(Object renderContext)` method) of an application is responsible for the visualisation (the rendering) of the XML document to the screen. In most Java based applications this is a class derived from `javax.swing.JComponent` or `java.awt.Component` and implements a `repaint()` method. In order to be compatible with this implementation of CEFX, the class representing the application's render context can be of any type but must

provide a `repaint()` method. The `CEFXDOMAdapter`'s `refresh()` method is called by the `CEFXController` when a remote operation is executed and the application should visualise the change to the user. The `CEFXDOMAdapter` in turn calls the `repaint()` method of the application's render context. If an application does not provide any such kind of render context, it has to take care of the visualisation of document changes by itself.

The method `setCollaborationReady(...)` is used to set the state of the framework to either "ready for collaboration" or "not ready for collaboration". The framework is set to "ready for collaboration" when all necessary initialisation has been performed and the collaboration can begin.

The method `closeSession()` is used to notify the framework that the client is leaving an editing session. In that case the `CEFXDOMAdapter` will close any open connections to the server and the other clients and sets the framework state to "not ready for collaboration". Thenceforth all incoming remote operations are ignored. This method is usually called when the editing application is closed.

6.1.2.1. Locking of *nodes*

The method `lock(Node node)` is used to lock a certain document node and thus prevent other users from editing that node until the node is unlocked again by using the `unlock(Node node)` method. Locking of nodes is realized by marking a node as locked by a certain client. The framework checks each node for this "lock marker" and prevents changing or deleting a node that has been marked as locked by another client other than the one trying to execute an operation on it. This checking is done at each client site whenever the client tries to modify the document using one of the above `CEFXDOMAdapter`'s methods corresponding to the DOM API. In order to mark a node as locked, the `CEFXDOMAdapter` creates an update operation that adds the attribute `CEFXLOCKED` to the node that is to be locked. The `CEFXLOCKED` attribute carries the client ID of the locking client as value. For example if the client with the `ID=1` locks a node the attribute `CEFXLOCKED=1` will be added to the node.

Unlocking of a node can only be performed by the client that locked it, all other clients would not be able to modify the `CEFXLOCKED` attribute. As mentioned above, the

`unlock(Node node)` method is used for this purpose. This method removes the `CEFXLOCKED` attribute from the specified node and thus readmits modifications from any other client within the editing session.

6.1.3. The client package

The client package contains the `CEFXClient` interface and its implementing class `CEFXClientImpl`.

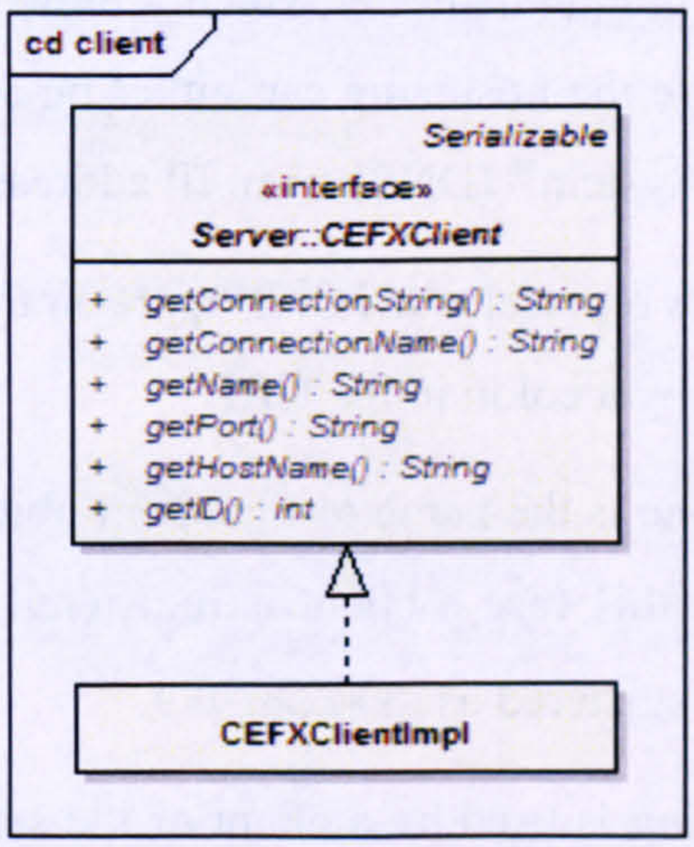

Figure 6.5: Client package classes

The `CEFXClient` is used to identify each client within an editing session and provide all other clients and the server with the information that is necessary for connecting to it. A `CEFXClient` object is transmitted over the network. Thus it must implement the `java.io.Serializable`[56] interface, allowing it to serialize a client object into a byte stream, ready for network transport.

The `CEFXClient` interface defines the following methods:

- `String getConnectionString();`
- `String getConnectionName();`
- `String getName();`
- `String getPort();`

[56] Serialization of objects is a standard feature of the Java runtime. In order to enable serialization, each serializable class has to be marked as such by implementing the java.io.Serializable interface.

- `String getHostName();`
- `String getID();`

The method `getConnectionString()` returns a String object containing a Unified Resource Identifier (URI). The URI[57] is a compact string of characters used to identify or name a resource over a network. The `CEFXClient` URI consists of the hostname, the port and the connection name. Below is a typical example of this:

```
//10.21.0.31:3451/CEFXClient
```

A hostname is the unique name by which a network attached device is known on a network. In this case the hostname can either be an internet hostname as defined by the Domain Name System[58] (DNS) or an IP address.

The port in this case represents a TCP/IP port on the clients machine and is separated from the hostname by a colon in the URI.

The connection name is the name of the client object as it is registered with the RMI registry service. In this case a client is registered as `CEFXClient` with the RMI registry. A server is registered as `CEFXServer`.

The connection string is used by a client or the server to connect via the network to the other clients. The methods `getConnectionName()`, `getPort()` and `getHostName()` are used to retrieve the information as indicated by the methods' names. The method `getName()` returns a String object containing the name of the client as it is defined at each client site. The first client in an editing session for example is named `Client1`, the second client is called `Client2` and so on. The method `getID()` returns a String object containing the unique id of the client. A clients name, id, hostname, port and connection name are configured in the CEFX network properties file when installing CEFX on a client or server computer and are retrieved from it when the `CEFXClient` is initialised. The content of the network property file and the installation of CEFX is further explained in chapter 7.3.

---

[57] For a definition of URI see: http://en.wikipedia.org/wiki/URI , retrieved October 30, 2007

[58] For more information on DNS see: http://en.wikipedia.org/wiki/Domain_Name_System , retrieved October 30, 2007

6.1.4. The client.net package

The `client.net` package contains all classes and interfaces that are used for handling the networking part of the CEFX client.

The `NetworkControllerImpl` class implements the `NetworkController` interface. It is composed of an `OutgoingClientConnectionHandler`, an `OutgoingServerConnectionHandler` and a `ClientConnection`. Additionally it is responsible for the client side session handling and thus owns a `CEFXSession` object (implemented by the class `CEFXSessionImpl`), which is retrieved from the server at the beginning of a session. Figure 6.6 shows the classes of the `client.net` package.

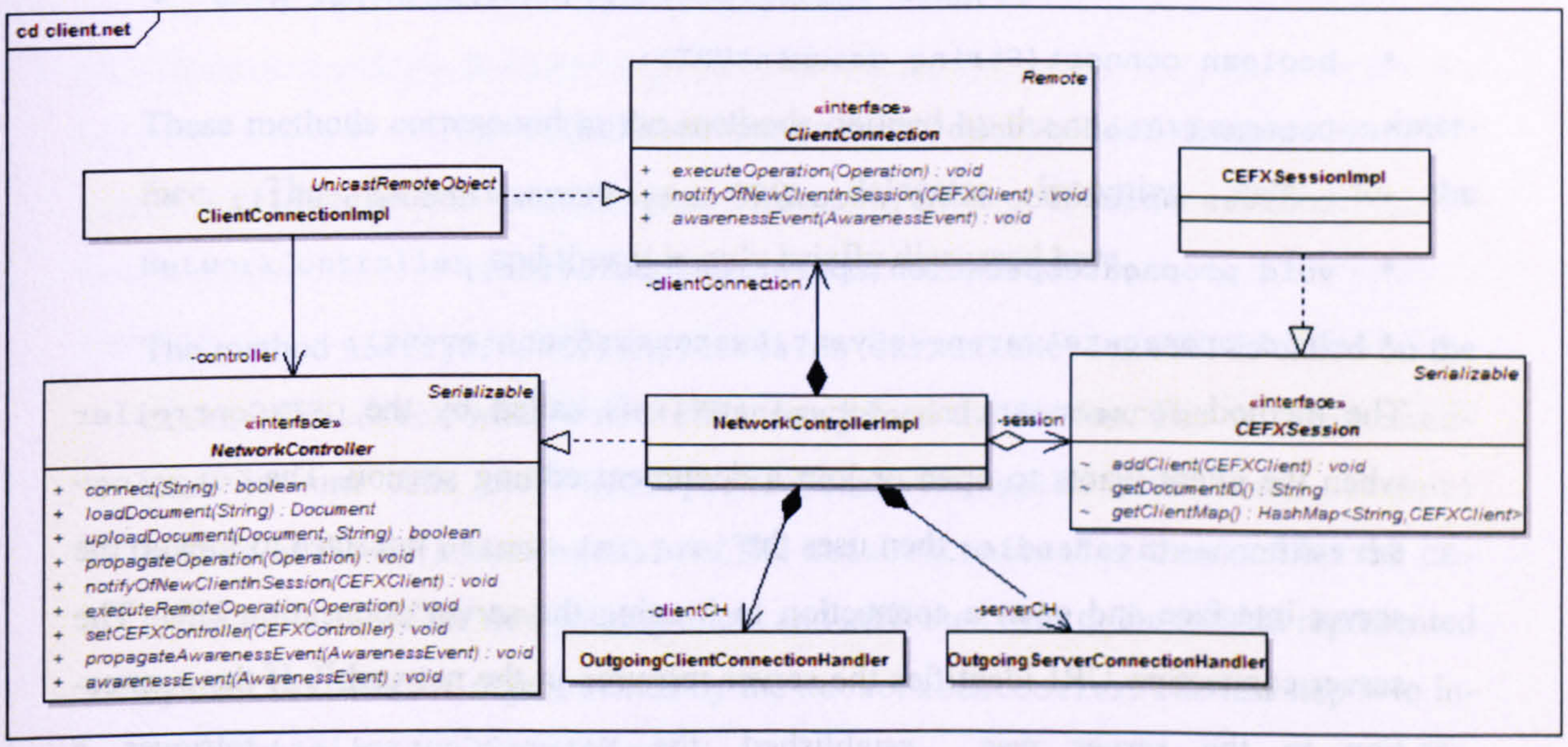

Figure 6.6: Classes of the client.net package

The `OutgoingClientConnectionHandler` is responsible for connecting to the other clients in a session and calling their corresponding client interface methods (using RMI). For example if the client executed an operation, the `OutgoingClientConnectionHandler` will connect to each other client in the session and call the `executeOperation(Operation o)` method of it. The `OutgoingServerConnectionHandler` is responsible for connecting to the server and propagating operations to it. The `ClientConnection` interface is implemented by the `ClientConnection-`

Impl class and defines the methods that are provided to the other clients and the server. In the following, the existing interface methods of the client.net package are discussed in detail.

6.1.4.1. The NetworkController *interface*

The NetworkController interface defines "outgoing" and "incoming" methods. "Outgoing" methods are those methods used by the CEFXController in order to send information to other clients or the server. "Incoming" methods are methods that are indirectly called by either another client or the server. They are called indirectly because all calls to a client are invoked on the ClientConnection interface which in turn calls the NetworkController methods. The "outgoing" methods are:

- `boolean connect(String documentURI);`
- `Document loadDocument(String documentURI);`
- `boolean uploadDocument(Document doc, String documentURI);`
- `void propagateOperation(Operation operation);`
- `void propagateAwarenessEvent(AwarenessEvent event);`

The method connect(String documentURI) is called by the CEFXController when the client wants to open or join a document editing session. The OutgoingServerConnectionHandler then uses the java.rmi.Naming interface to lookup the server interface and open a connection to it using the server connection URI. The server connection URI identifies the server resource in the network[59]. If the connection to the server was established, the NetworkController retrieves a ServerConnection object from the OutgoingServerConnectionHandler and calls its connect(CEFXClient client, String documentURI) method.

In order to retrieve a document from the server or upload a document to it, the methods loadDocument(...) and uploadDocument(...) are used by the CEFXController. When calling these methods, the NetworkController calls the corresponding methods of the ServerConnection interface (discussed in chapter 6.2.2).

---

[59] For more information on RMI see: http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html , retrieved October 30, 2007

In order to propagate operations or awareness events, the `NetworkController` interface defines the methods `propagateOperation(...)` and `propagateAwarenessEvent(...)`. Operations are propagated to both, the server and all clients in a session. Awareness events are only propagated to the clients. The propagation of operations and awareness events to the clients is handled by the `OutgoingClientConnectionHandler` class. The propagation of operations to the server is handled by the `OutgoingServerConnectionHandler` class. The "incoming" methods of the NetworkController interface are:

- `void notifyOfNewClientInSession(CEFXClient client);`
- `void executeRemoteOperation(Operation operation);`
- `void awarenessEvent(AwarenessEvent event);`

These methods correspond to the methods defined by the `ClientConnection` interface. The `ClientConnection` only delegates incoming calls to the `NetworkController` and thus it is only briefly discussed here.

The method `notifyOfNewClientInSession(CEFXClient client)` is called on the `ClientConnection` when a new client has joined the session. The `ClientConnection` in turn calls the `notifyOfNewClientInSession(CEFXClient client)` method of the `NetworkController`. The `NetworkController` then notifies the `CEFXController` of the new joining client and adds the client to the session represented by the `CEFXSession` object owned by the `NetworkController`. The last step is to inform the `OutgoingClientConnectionHandler` of the new client. The `OutgoingClientConnectionHandler` stores a reference to each client's `ClientConnection` interface and uses the `java.rmi.Naming` interface to lookup the remote interface (`ClientConnection`) of the new joined client. As mentioned before, the `CEFXClient` interface (provided as argument) contains all information necessary for a RMI lookup.

The method `executeRemoteOperation(Operation operation)` of the `NetworkController` is called by the `ClientConnection` when another client calls its `executeOperation(Operation operation)` method. The `NetworkController` delegates the call to the `CEFXController`.

When the method `awarenessEvent(AwarenessEvent event)` of the `ClientConnection` interface is called, the call is delegated to the corresponding method of the `NetworkController`. The `NetworkController` in turn delegates the call to the `AwarenessController`.

The last method of the NetworkController interface is:

- `void setCEFXController(CEFXController controller);`

This method is called, when the `CEFXController` initialises the `NetworkController` and is used to provide it with a reference to the `CEFXController`.

6.1.4.2. The CEFXSession *interface*

A `CEFXSession` represents a concurrent editing session and contains information on the different clients and the document that is edited. A `CEFXSession` object is created by the server and transmitted over the network to all clients taking part in the editing session. Each client thereby retrieves a copy of the server's `CEFXSession` object when it connects to the server (discussed in chapter 6.2). The `CEFXSession` interface is implemented by the `CEFXSessionImpl` class and defines the following methods:

- `void addClient(CEFXClient client);`
- `String getDocumentID();`
- `HashMap<String,CEFXClient> getClientMap();`

The method `addClient(CEFXClient client)` is called by the server when a client connects to it. It adds the new client to the session's map of clients. The map of clients (`clientMap`) contains the name and a `CEFXClient` object identifying each client.

The method `getDocumentID()` returns the document's URI which is a path to the document relative to its location in the local document repository. This URI is used for the identification of a document and thus is called document id in this context. The server uses this document id to check if a client requests to connect to an existing session or to a new one.

The method `getClientMap()` returns the session's map of clients. This method is

called by the NetworkController in order to add a new client to its local CEFXSession object. This is done in the notifyOfNewClientInSession(CEFXClient client) method of the NetworkController.

6.1.5. The registry and extension packages

As mentioned before, the ExtensionRegistry contains information on the plug-in components of CEFX. The registry package contains the ExtensionRegistry class which is initialised by the CEFXController. On initialisation, the ExtensionRegistry loads the CEFX configuration XML file (cefx.xml) which contains the extension points information as discussed in chapter 5.3.1.1. The information in the CEFX configuration XML file is loaded into a set of Java objects using the Java Architecture for XML Binding (JAXB)[60]. The classes that represent this information are located in the extension package. These classes were generated using the JAXB XML Schema parser and class generator on basis of the XML Schema file cefxextension.xsd. The JAXB API provides methods for marshalling (writing Java objects to an XML file) and unmarshalling (reading Java objects from an XML file). The ExtensionRegistry unmarshalls the content of the cefx.xml file and provides the following method to retrieve the required extension point information:

- `Object getExtensionPointConfigurationItem(String extensionPoint);`

This method is used by the CEFXController when it initialises the CEFX plug-in components. The extensionPoint argument identifies the requested extension point information. The object that is returned here is – depending on the given extensionPoint argument – of a type of one of the following classes from the extension package:

- de.hdm.cefx.extension.AwarenessExtension
- de.hdm.cefx.extension.NetworkController
- de.hdm.cefx.extension.ConcurrencyController
- de.hdm.cefx.extension.ConflictResolutionModule

---

[60] For more information on JAXB see: https://jaxb.dev.java.net/ , retrieved October 30, 2007

The `AwarenessExtension` class contains information on the `AwarenessController` plug-in and the `AwarenessWidgets`. It provides the following methods in order to retrieve that information:

- `AwarenessController getAwarenessController();`
- `AwarenessWidget getAwarenessWidget();`

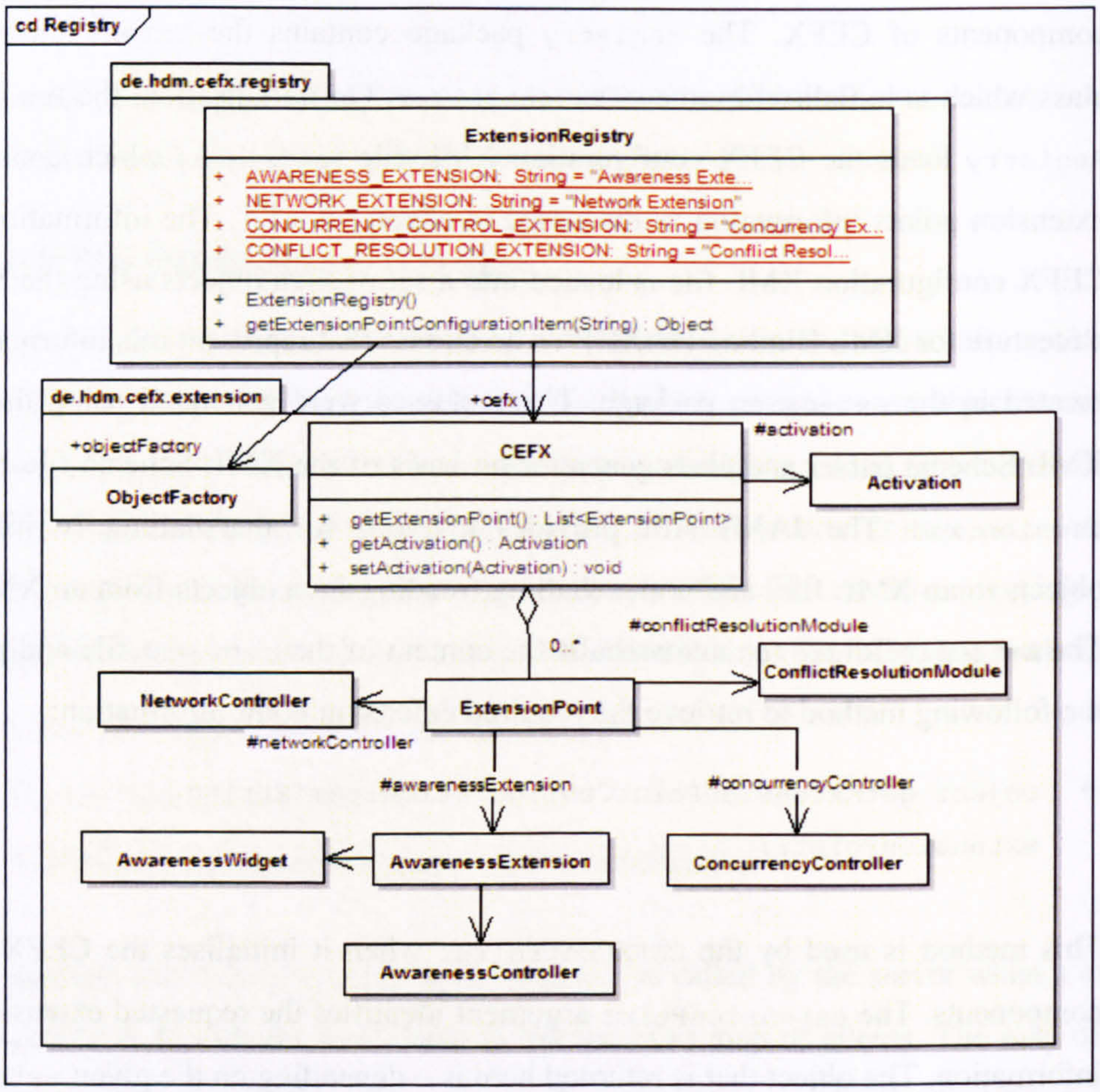

Figure 6.7: Classes in the registry and extension packages

The `AwarenessController` and the `AwarenessWidget` classes contain information on the Java class that should be loaded by CEFX as `AwarenessController` or `AwarenessWidget`. This information can be retrieved by using their `getClazz()` method. The `NetworkController`, the `ConcurrencyController` and the `ConflictResolutionModule` classes from the `extension` package also provide a

method `getClazz()` which is used by the `CEFXController` in order to instantiate the corresponding plug-in component. Figure 6.7 depicts the classes from the `registry` and the `extension` packages.

The `CEFX` class represents the root node of the CEFX configuration. It provides methods to retrieve the contained extension points and the list of activated plug-in components which is encapsulated in the `Activation` class.

If the CEFX configuration XML file does not contain all necessary information, or does not exist, the `ExtensionRegistry` creates a default configuration. This is done by using the methods of the `ObjectFactory` class. The `ObjectFactory` class is also generated by JAXB and provides a set of "create" methods (for example the method `createConcurrencyController()`) that can be used to instantiate objects of the above classes and fill them with the required information.

6.1.6. The util package

The `util` package contains two classes that were developed in order to facilitate using CEFX for an application programmer (`CEFXUtil` class) and to clearly separate the Aspect Oriented Programming (AOP) source code from the Java source code (`Advice` class). The two classes are shown in figure 6.8.

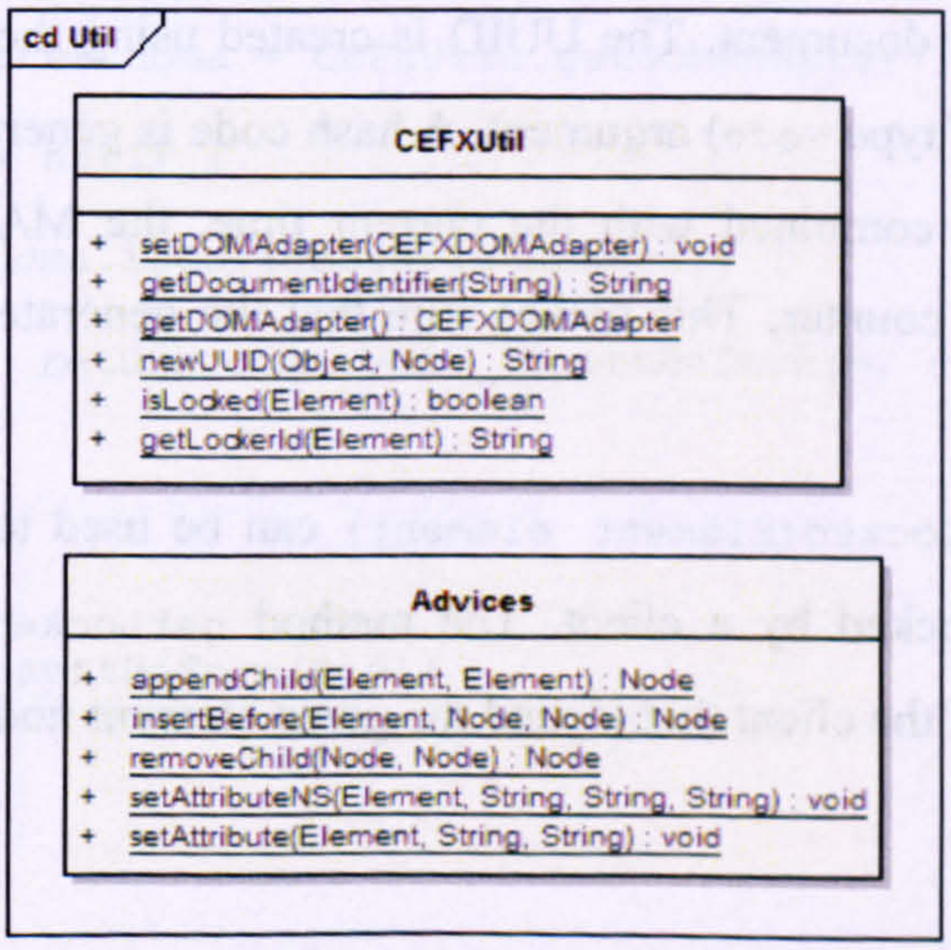

Figure 6.8: Classes in the util package

The `CEFXUtil` class defines the following static methods:

- `void setDOMAdapter(CEFXDOMAdapter adapter);`
- `CEFXDOMAdapter getDOMAdapter();`
- `String getDocumentIdentifier(String documentURI);`
- `String newUUID(Object client, Node node);`
- `boolean isLocked(Element element);`
- `String getLockerId(Element element);`

The `CEFXDOMAdapter` uses the method `setDOMAdapter(CEFXDOMAdapter adapter)` to provide the `CEFXUtil` class with a reference to it. The method `getDOMAdapter()` can then be used by other classes to retrieve this reference.

The method `getDocumentIdentifier(String documentURI)` is a utility method that returns a client independent URI (as an object of the type `String`) for a given document path. The given `documentURI` argument can thereby contain a full path to a document file. The resulting URI will contain a relative path to the document starting from the local repository location. This method is used when a client connects to the server and wants to create or join an editing session for a certain document.

The method `newUUID(Object client, Node node)` is used by the `CEFXDOMAdapter` for creating a universally unique identifier (UUID) for a new node that is to be inserted into the document. The UUID is created using the client (of type `Object`) and the node (of type `Node`) argument. A hash code is generated on basis of these arguments and is combined with the current time, the MAC address of the client computer and a counter. This makes sure that the generated UUID is truly universally unique[61].

The method `isLocked(Element element)` can be used to check if the given element node is locked by a client. The method `getLockedId(Element element)` returns the ID of the client that locked the given element node, if any.

---

[61] For more information on UUIDs see: http://en.wikipedia.org/wiki/UUID , retrieved October 30, 2007

The Advices class defines the following static methods:

- `Node appendChild(Element parent, Element child);`
- `Node insertBefore(Element parent, Node currentChild, Node oldChild);`
- `Node removeChild(Node parent, Node child);`
- `void setAttributeNS(Element parent, String nameSpace, String attribute, String value);`
- `void setAttribute(Element parent, String attribute, String value);`

In each of these methods it is checked, that the framework is ready for collaboration by calling the `isCollaborationReady()` method of the `CEFXDOMAdapter`. If that is the case the call is forwarded to the corresponding method of the `CEFXDOMAdapter`. If the framework is not ready for collaboration (for example if it has not been properly initialised) the corresponding action is directly executed on the given parent Element node. The following source code example shows, how this is done in the case of a call to the `insertBefore(...)` method of the Advices class.

```
public static Node insertBefore(Element p, Node c, Node o)
{
      CEFXDOMAdapter doma = CEFXUtil.getDOMAdapter();
      if (doma != null) {
            if (doma.isCollaborationReady()) {
                  return doma.Node_insertBefore(p, c,o);
            }
      }
      return p.insertBefore(c,o);
}
```

6.1.7. The awareness package

CEFX provides the user with information on other user's actions. The classes that allow the integration of awareness mechanisms into a collaborative real-time editing system are located in the `awareness` package of CEFX. The main component is the `AwarenessController` (AC). The AC receives awareness information from own or other clients in the form of an `AwarenessEvent` object. It propagates the `AwarenessEvent` to the other clients or visualises the information with the help of an `AwarenessWidget`. CEFX provides default `AwarenessController` and `AwarenessWidget` implementations. The `DefaultWidget` class visualises information in the form of a small window that shows text messages to a user if, for example, another user inserted a new node into the shared document, or selected a node using the computer mouse. When editing a large document, where a user can only see a small part of it, this information can help to "be aware" of what other users are working on.

Figure 6.9 shows the classes and interfaces in the `awareness` and the `awareness.event` package.

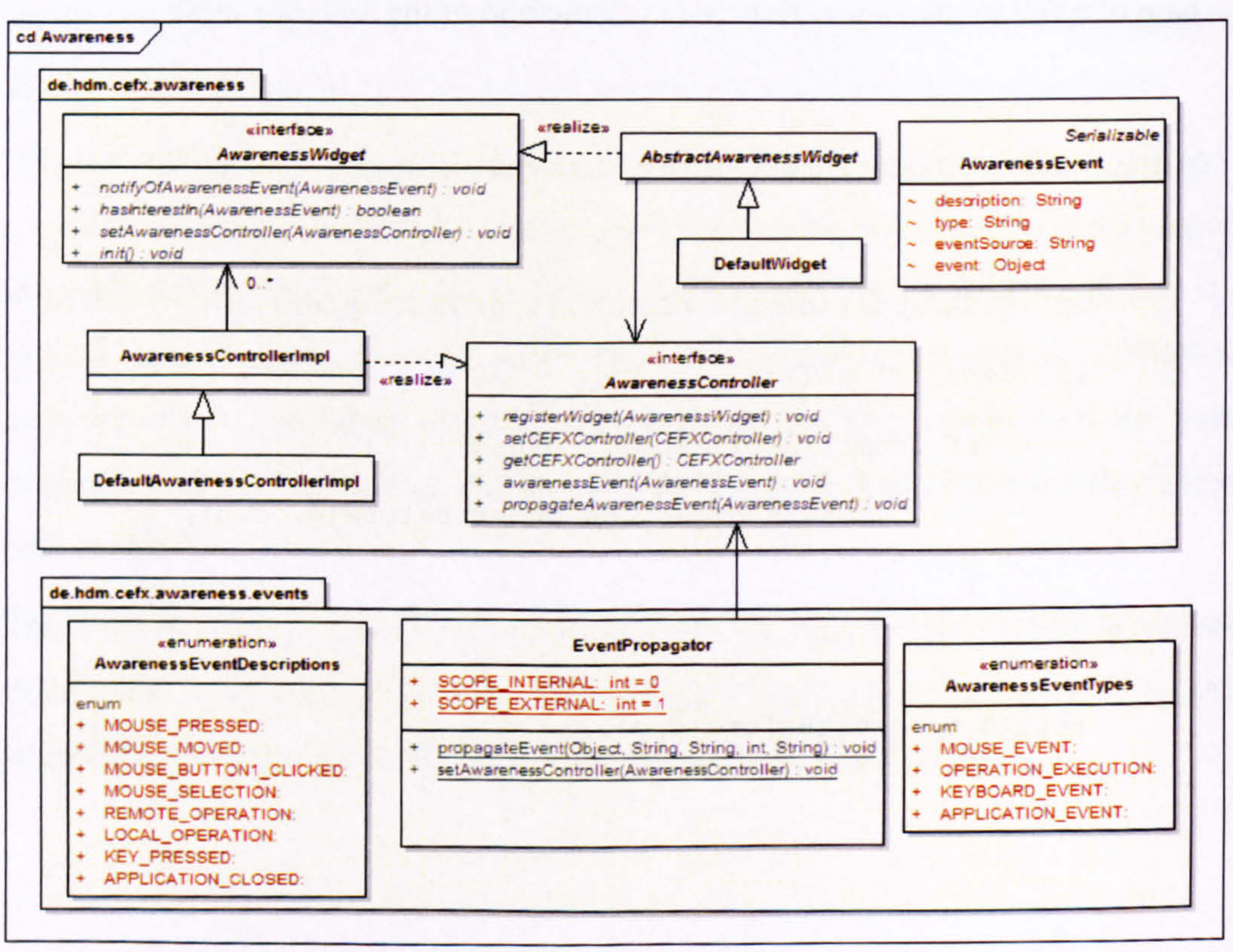

Figure 6.9: Classes in the awareness and awareness.event package

The AwarenessController interface is implemented by the AwarenessControllerImpl class. This class is the base class for AC extensions and the default AC provided by CEFX is implemented in the DefaultAwarenessControllerImpl class. The AwarenessControllerImpl is provided with references to the awareness widgets. The AwarenessWidget interface defines the methods that need to be implemented by an awareness widget. The class AbstractAwarenessWidget implements those methods and acts as the base class for awareness widget extensions such as the DefaultWidget class. The AwarenessEvent class carries awareness event information such as the type of an event, the event description, the source of the event (for example the clients name) and an event object. The event object can be of any type, it is only required that the awareness widget that is interested in that event knows how to deal with it.

The awareness.event package contains the EventPropagator class which is responsible for propagating events to the AC. The enumerations AwarenessEventTypes and AwarenessEventDescriptions define a set of possible event types and descriptions but CEFX is not limited to those. They are merely used to facilitate the implementation of awareness widgets.

In the following the AwarenessController and AwarenessWidget interface methods are discussed before explaining how events are propagated with the help of the EventPropagator class.

6.1.7.1. The AwarenessController *interface*

The following methods are defined by the AwarenessController interface:

- `void registerWidget(AwarenessWidget widget);`
- `void setCEFXController(CEFXController controller);`
- `CEFXController getCEFXController();`
- `void awarenessEvent(AwarenessEvent event);`
- `void propagateAwarenessEvent(AwarenessEvent event);`

As mentioned before, the AwarenessController is initialised by the CEFXController. The CEFXController also initialises the AwarnessWidgets and registers them

with the AwarenessController. This is done by using the registerWidget(...) method. The CEFXController provides the AwarenessController with a reference to it by calling the setCEFXController(...) method. The AwarnessController uses this reference to retrieve a reference to the NetworkController in order to send awareness events over the network. The getController(...) method returns the reference to the CEFXController. When an event is to be forwarded to an awareness widget, the awarenessEvent(...) method of the AwarenessController must be called. The method propagateAwarenessEvent(...) propagates an event to the other clients in a session. Before an event is visualised or propagated, it is checked, if any widget exists that is interested in such an event. If no widget is interested, the event is dropped.

6.1.7.2. The *AwarenessWidget* interface

The AwarnessWidget interface defines the following methods:

- `boolean hasInterestIn(AwarenessEvent event);`
- `void notifyOfAwarenessEvent(AwarenessEvent event);`
- `void setAwarenessController(AwarenessController ac);`
- `void init();`

In order to check if a widget is interested in a certain event, the method hasInterestIn(...) is used. If the widget is interested, the AC notifies it of the event by calling the notifyOfAwarenessEvent(...) method. The widget will then visualise the event in some way to the user.

When a widget is registered with the AC, it is provided with a reference to it by a call to the setAwarenessController(...) method. The AC initialises the widget by calling its init() method. When initialised, a widget usually displays some sort of window or dialogue to the user.

The DefaultWidget implementation of CEFX opens a small window with a text box that shows a message. This small window is independent of the editing application and can be closed or minimized, if it is not needed. The figure below shows a screen shot of the default widget.

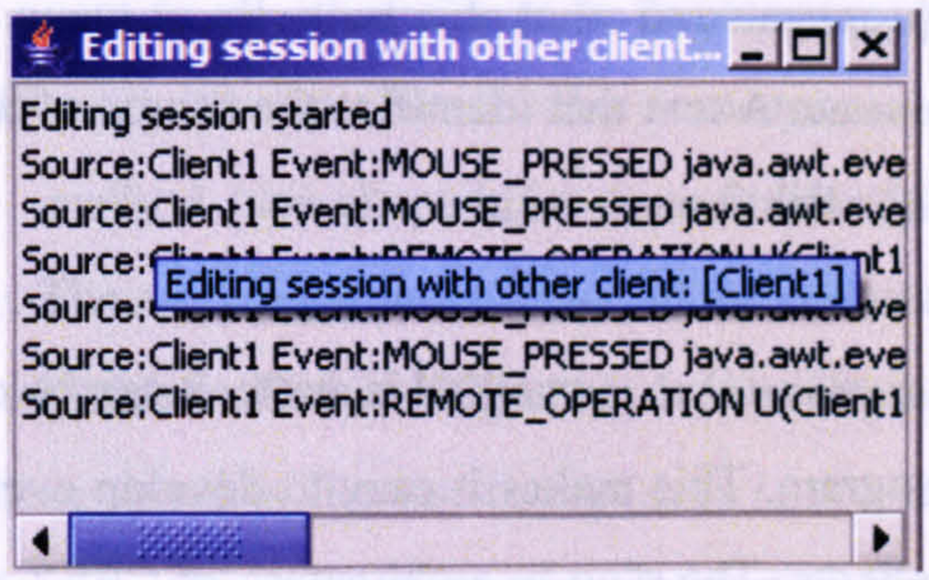

Figure 6.10: The DefaultWidget visualises the awareness events as text messages

6.1.7.3. Propagating *events*

As mentioned before the `EventPropagator` class is used to propagate the awareness events to the AC. It defines the following static methods:

- `void propagateEvent(Object event, String type, String description, int scope, String source);`
- `void setAwarenessController(AwarenessController ac);`

The AC provides the `EventPropagator` with a reference to it by calling the `setAwarenessController(...)` method. The `EventPropagater` uses this reference to forward events to the AC. In order to propagate an awareness event, the `propagateEvent(...)` method is used. The `propagateEvent(...)` method requires the following arguments to be passed to it:

- **`Object event`**. This can be an object of any type and represents the original event that occurred. For example a mouse event or a key event.
- **`String type`**. A text that identifies the type of event. For example "`MOUSE_EVENT`" or "`KEYBOARD_EVENT`". The `type` argument will be used to find the awareness widgets that are interested in this event.
- **`String description`**. A text that describes the event in further detail.
- **`int scope`**. The scope of the event. Two scopes exist, the internal and the external scope. The internal scope is for events that should not be propagated to

other clients but visualised to the user. The external scope is for events that should be propagated to other clients.

- `String source`. A text that identifies the source of the event. This could be, for example, the client's name or id.

As the `propagateEvent(...)` method is static, it can be called from nearly any context within a program. This makes it easy to develop event listeners that forward an event to other clients. The plug-in architecture of CEFX makes it easy to create and integrate new awareness widgets that react to those events and visualise them.

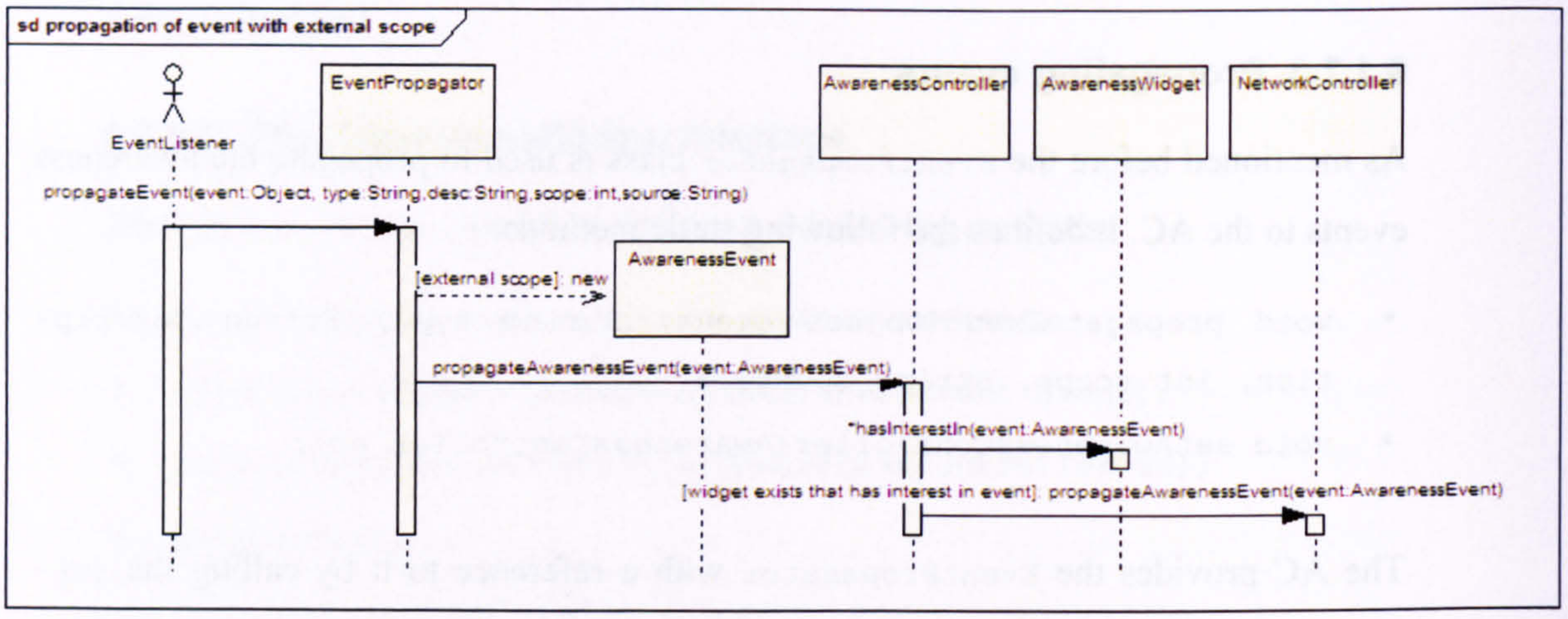

Figure 6.11: Propagation of an awareness event with external scope

Figure 6.11 shows a scenario of event propagation. The `EventListener` in the sequence diagram is for example a mouse listener (e.g. `java.awt.event.MouseListener`) that reacts on computer mouse clicks. If a user clicks for example on a certain object, the mouse listener calls the static `propagateEvent(...)` method of the `EventPropagator`. The given scope in this case is external, meaning that the event will be propagated to all other clients in the session. The `EventPropagater` creates a new `AwarenessEvent` object and passes it to the `AwarenessController` using the `propagateAwarenessEvent(...)` method. The `AwarenessController` then checks each registered `AwarenessWidget` if it is interested in the event. If an interested widget is found, the `AwarenessController` calls

the `propagateAwarenessEvent(...)` method of the `NetworkController` which in turn will propagate the event to all other clients in the session. At the other client's site, the `NetworkController` receives the awareness event via its `awarenessEvent(...)` method (see chapter 6.1.4) and directly forwards it to the `AwarenessController`. The `AwarnessController`, in the context of its own `awareness-Event(...)` method, tries to find a widget that is interested in that event. Thus the `AwarenessController` invokes the `hasInterestIn(...)` method on each registered `AwarenessWidget`. If a widget is interested in the event, the `notifiyOfAwarenessEvent(...)` method of the widget is called and the widget then visually alerts the user to the event.

Figure 6.12 shows the described scenario of an incoming remote awareness event.

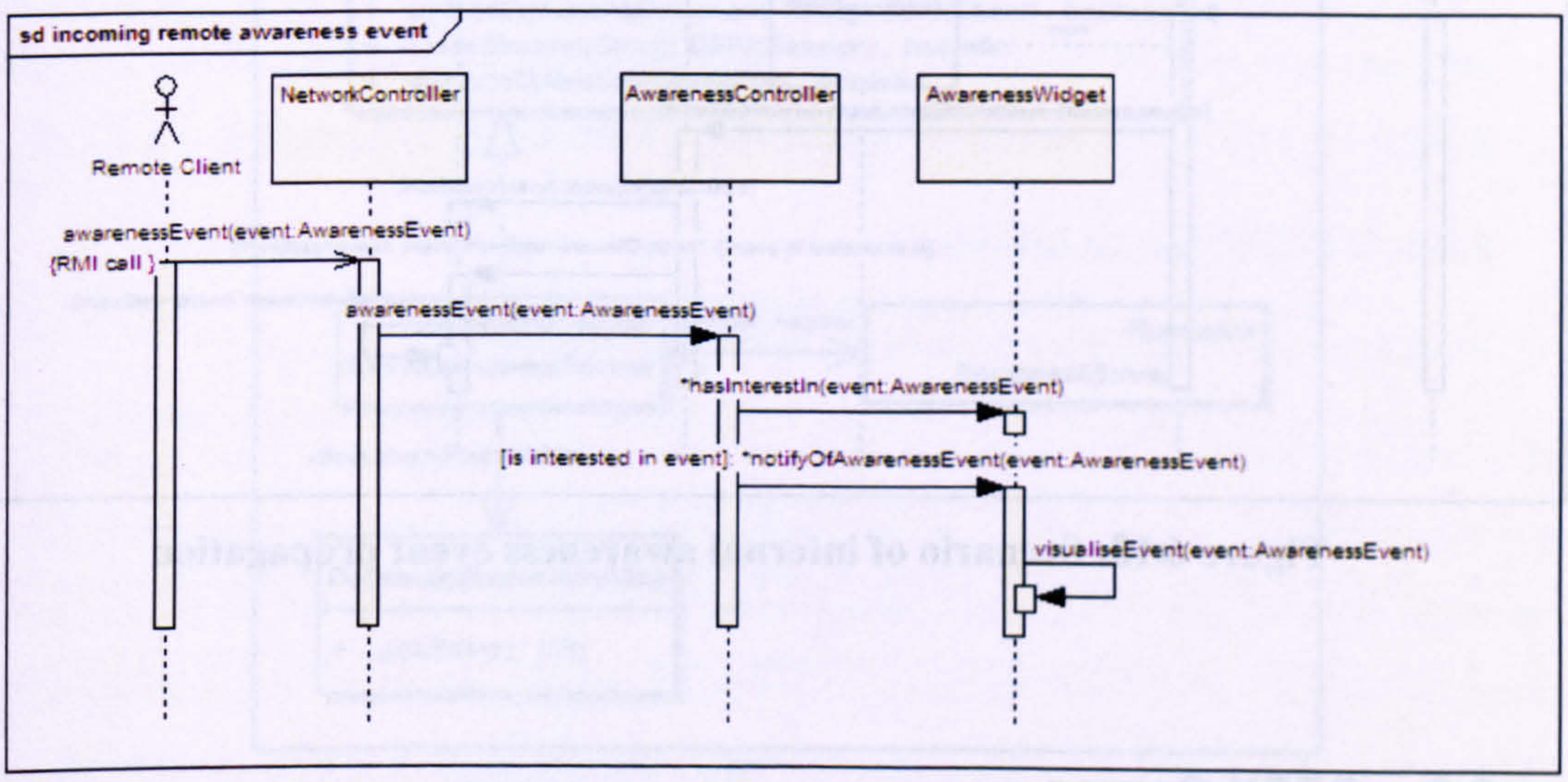

Figure 6.12: Scenario of an incoming remote awareness event

In the case of awareness events that should not be propagated to other clients but visualised (via a widget) to the user, the internal scope for event propagation exists. The internal propagation of events may make sense, for example, to notify the user of a new user joining the session. It can also be used to inform the user of conflicts of locked nodes or - as it was used in this implementation of CEFX – of the execution of a remote operation.

Figure 6.13 shows a scenario of an internal event propagation. This scenario is similar to the scenario in figure 6.12 except that the awareness event is not received by the

NetworkController. The EventListener (this could be any class being triggered by an event) calls the EventPropagator's propagateAwarenessEvent(...) method with the scope argument set to "internal". Thus the EventPropagator this time calls the awarenessEvent(...) method of the AwarnessController instead of the propagateAwarenessEvent(...) method. The AwarnessController in turn checks the registered widgets if they are interested in the event. It calls the notifyOfAwarenessEvent(...) method on each widget that has an interest in the awareness event. The widgets then visualise the event.

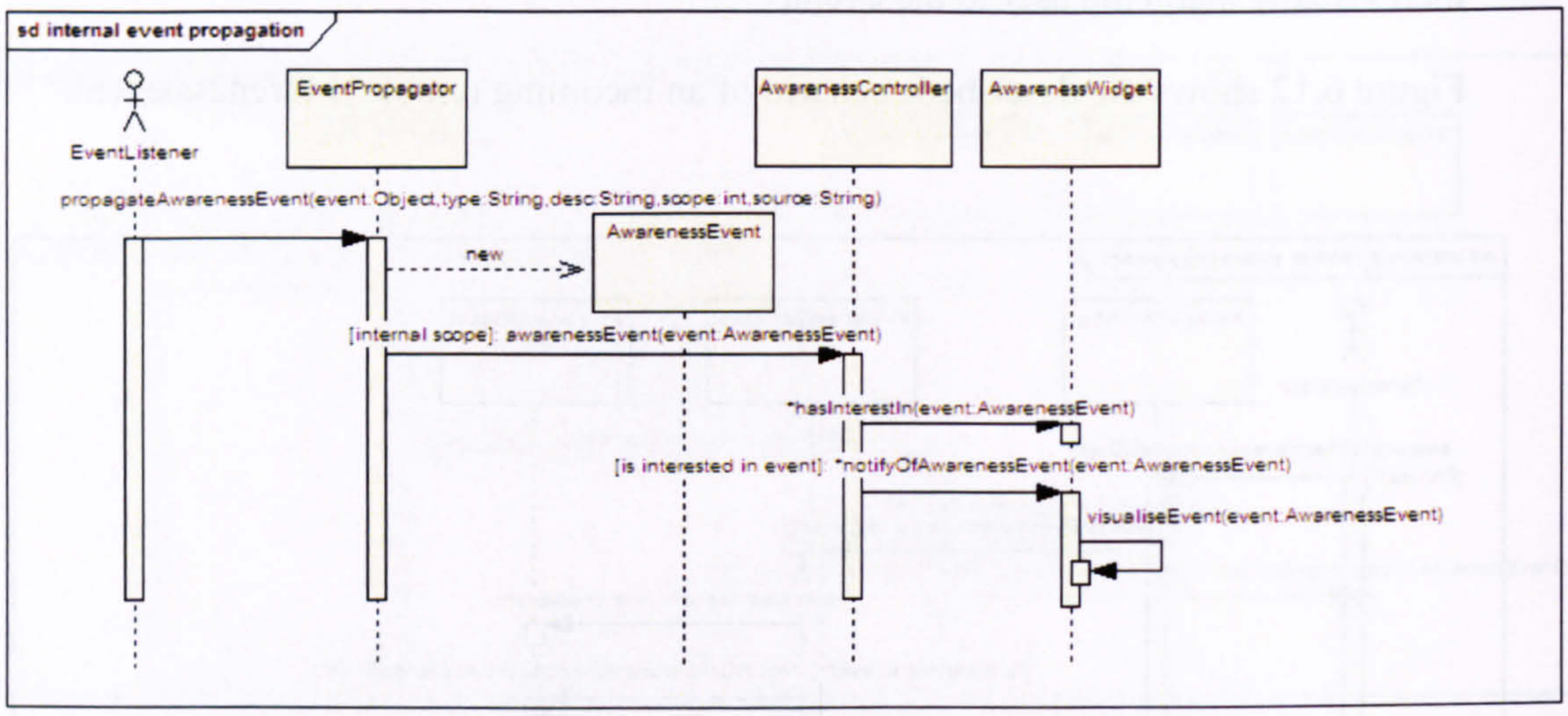

Figure 6.13: Scenario of internal awareness event propagation

6.2. CEFX Server

The CEFX server was implemented as a stand-alone application and is independent of the CEFX client but relies on the same classes as the client for concurrency control. The server keeps a local copy of the shared document in an editing session and executes the same operations as each client. For concurrency control the ConcurrencyController and ConflictResolutionProvider classes are provided to the server in a JAR archive and linked into the server's class path.

In contrast to the client, the server does not require the DOMAdapter or the AwarenessController and AwarenessWidget components as it does not connect to an

editing application and does not have any user interface.

The server's source code is structured into a base package (`de.hdm.cefx.server`), a networking package (`de.hdm.cefx.server.net`) and a utility package (`de.hdm.cefx.server.util`).

The following sections discuss these packages and their contained components.

6.2.1. The CEFX server base package

The server's base package contains the classes illustrated in figure 6.14.

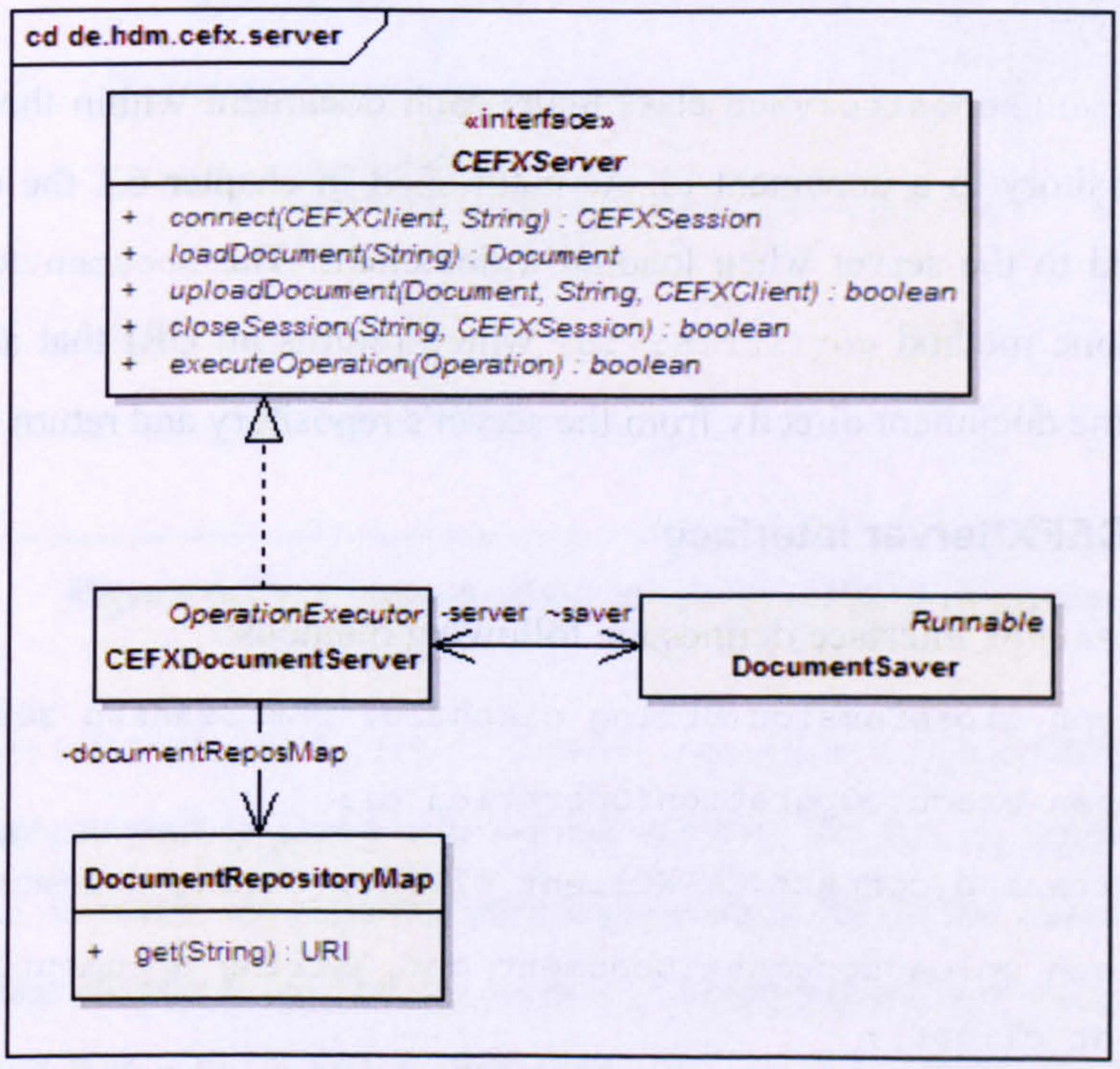

Figure 6.14: CEFX server classes in package de.hdm.cefx.server

The `CEFXDocumentServer` class implements the `CEFXServer` interface and is the main class of the server. It also implements the `OperationExecutor` interface and thus acts - from the perspective of the `ConcurrencyController` – as client. The `CEFXServer` component is (similar to the `CEFXController` component of the client) responsible for maintaining a sessions with each client that is connected to it. It is also responsible for providing each new connected client with a copy of the shared document.

The server stores the current version of the edited document in its local repository. This allows it to store the changes of the edited document, even if the server is stopped or the editing session ends. The next time a client opens a session for the same document, it will contain the same state as when the previous session ended. Storing the document periodically is done by the `DocumentSaver` class. The `DocumentSaver` was implemented as an independent thread (using the `java.lang.Runnable` interface) that stores the document after a certain time (in this case every 30 seconds). The name of the server's local document repository is `ServerTempRepository` which is located in the CEFX installation directory (see chapter 7.3).

The `DocumentRepositoryMap` class maps each document within the server's document repository to a document id. As mentioned in chapter 6.1 the document id is transmitted to the server when loading a document. The `DocumentRepositoryMap` only has one method `get(String id)` which returns an URI that allows to locate and load the document directly from the server's repository and return it to a client.

6.2.1.1. CEFXServer interface

The `CEFXServer` interface defines the following methods:

- `boolean closeSession(String clientId, CEFXSession session);`
- `boolean executeOperation(Operation o);`
- `CEFXSession connect(CEFXClient client, String documentURI);`
- `boolean uploadDocument(Document doc, String documentURI, CEFX-Client client);`
- `Document loadDocument(String documentURI);`

The `closeSession(String clientId, CEFXSession session)` method is called when a client wants to disconnect from the current editing session. The server then removes the calling client from the session. The method `executeOperation(Operation o)` is called when a remote operation arrives at the server site. The server delegates the execution of the operation to the `ConcurrencyController`.

The activity diagram in figure 6.15 shows the program flow at the server site when a client connects to the server.

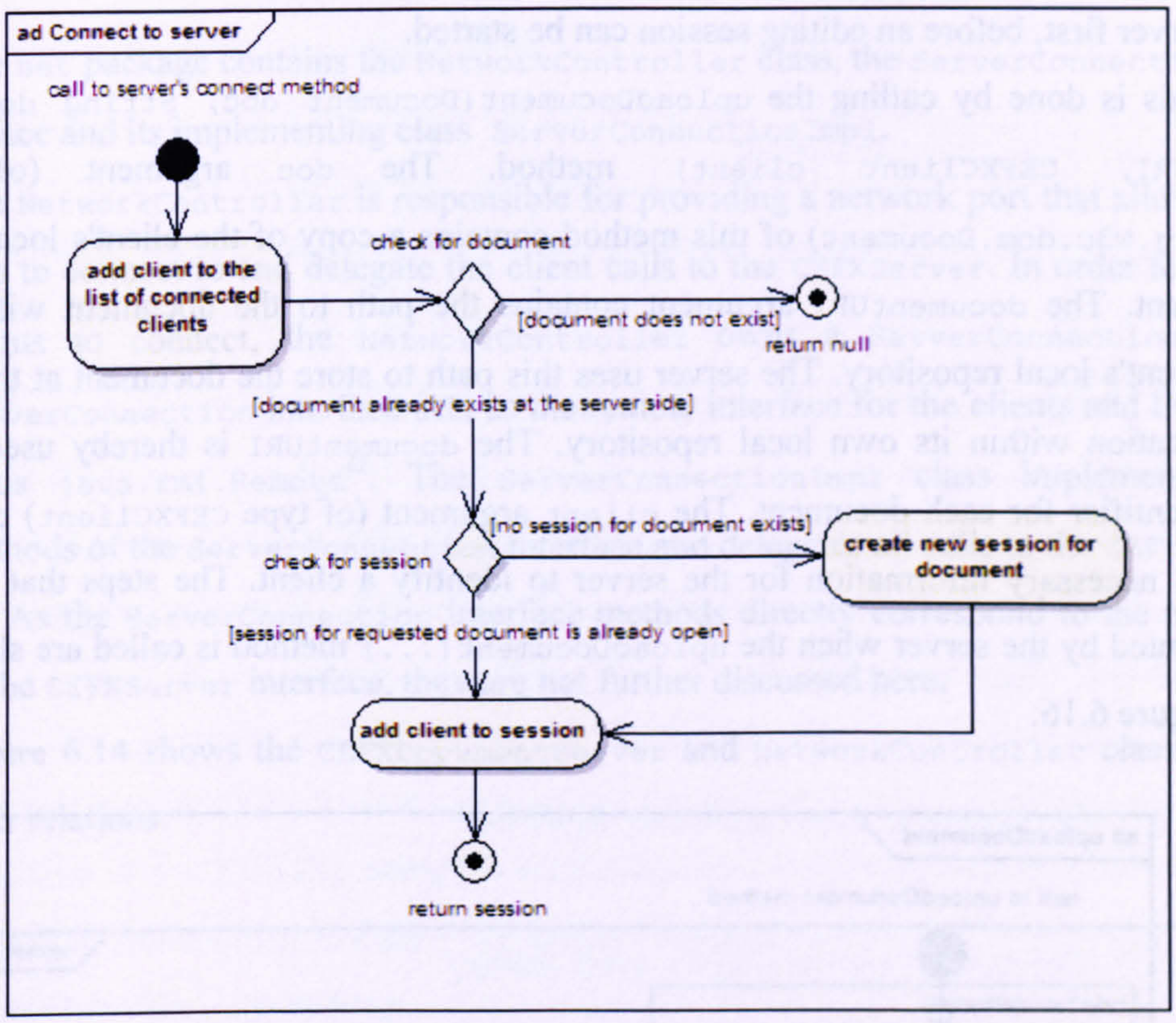

Figure 6.15: Connect client to server activity diagram

The `connect(CEFXClient client, String documentURI)` method is called when a client wants to start or join a new editing session. The `client` argument (of type `CEFXClient`) contains all required information about the client, such as name and identifier (see chapter 6.1). The `documentURI` argument (of type `String`) identifies the document that is to be edited. After the `connect(...)` method has been called, the server adds the client to his list of connected clients. Then the server checks if the requested document has already been stored in a previous session at the server site (in the server's local document repository). If the document is found, the server checks if a session for the document already exist, which is the case if another client has started a session before. If no session exists, the server creates a new session. Then the client is added to that session and the server returns the session object to the client. If the server cannot find the requested document in its local repository, a null pointer instead of a session object is returned to the client. This informs the client of the fact that the server does not have the document so it cannot start a session and

connecting the client fails. In this case the client needs to upload the document to the server first, before an editing session can be started.

This is done by calling the `uploadDocument(Document doc, String documentURI, CEFXClient client)` method. The `doc` argument (of type `org.w3c.dom.Document`) of this method contains a copy of the client's local document. The `documentURI` argument contains the path to the document within the client's local repository. The server uses this path to store the document at the same location within its own local repository. The `documentURI` is thereby used as an identifier for each document. The `client` argument (of type `CEFXClient`) contains all necessary information for the server to identify a client. The steps that are executed by the server when the `uploadDocument(...)` method is called are shown in figure 6.16.

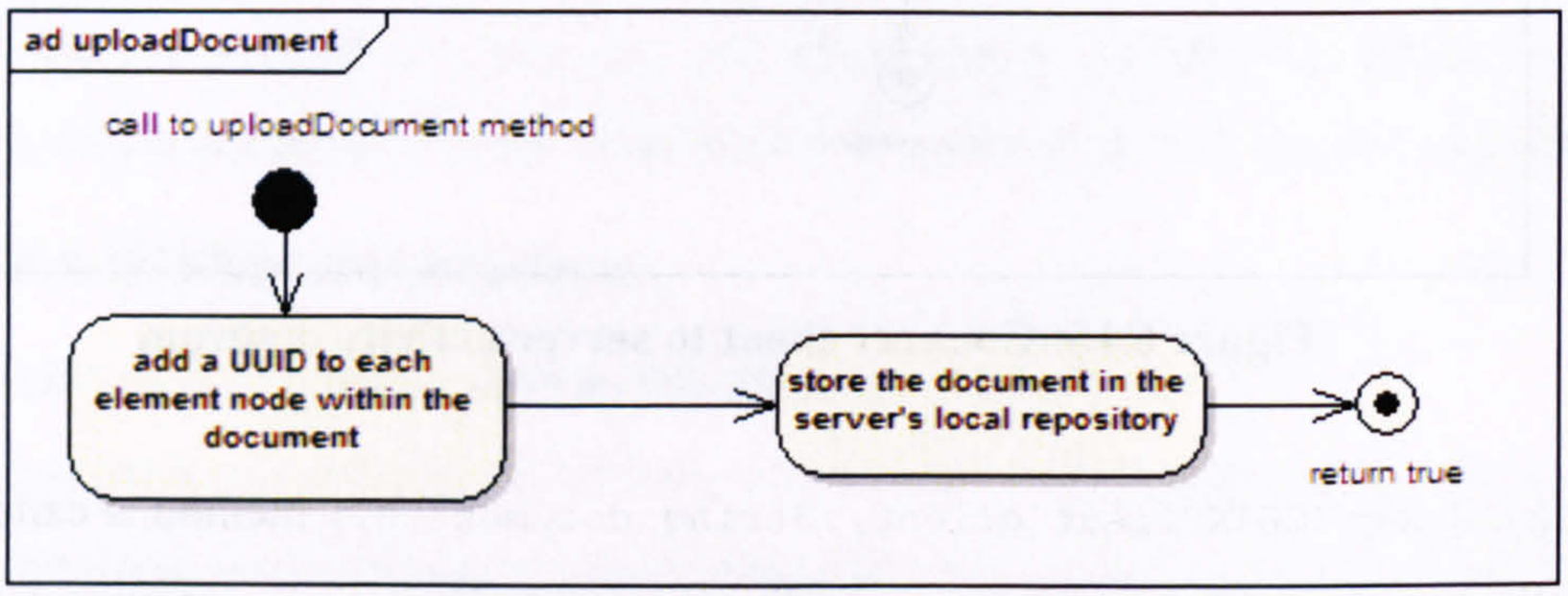

Figure 6.16: Uploading a document activity diagram

The server prepares the document for concurrent editing, by adding UUIDs to each element node. This allows easy identification of each node within the document. After this is done, the document is stored in the server's local repository in order to be able to send it to the next client connecting to the session.

After the client has uploaded the document, it connects to the server again. This time, connecting will not fail. The next step for the client is to retrieve the processed document (now containing UUIDs) from the server. This is done by calling the `loadDocument(String documentURI)` method of the server interface. When this method is called the server locates the document within its local repository by using the given `documentURI` argument. If the document is found, the server returns a copy of it to the client (as object of the type `org.w3c.dom.Document`).

6.2.2. The net package

The `net` package contains the `NetworkController` class, the `ServerConnection` interface and its implementing class `ServerConnectionImpl`.

The `NetworkController` is responsible for providing a network port that allows clients to connect to and delegate the client calls to the `CEFXServer`. In order to allow clients to connect, the `NetworkController` owns a `ServerConnection`. The `ServerConnection` interface acts as the remote interface for the clients and thus extends `java.rmi.Remote`[62]. The `ServerConnectionImpl` class implements the methods of the `ServerConnection` interface and delegates all calls to the `CEFXServer`. As the `ServerConnection` interface methods directly correspond to the method of the `CEFXServer` interface, they are not further discussed here.

Figure 6.14 shows the `CEFXDocumentServer` and `NetworkController` classes and their relations.

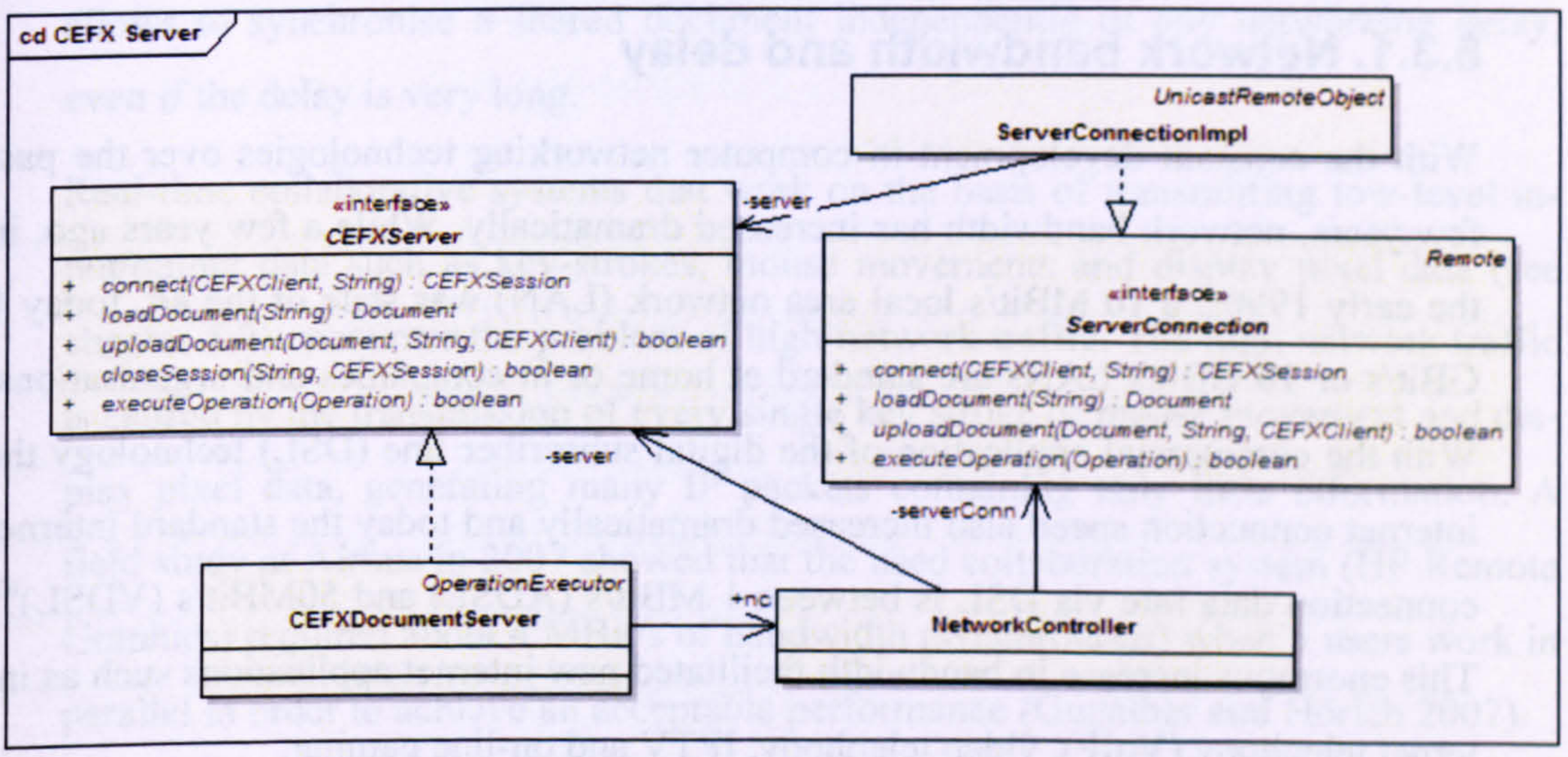

Figure 6.17: CEFX server classes

When the server is started, the `NetworkController` is initialised and subsequently initialises the `ServerConnectionImpl`. It then binds the `ServerConnection` with the RMI registry in order allow the clients to look up the server interface and call the

[62] Java Remote Method Invocation (RMI) is used in this implementation of the NetworkController. For more information on RMI see: http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp , retrieved October 30, 2007

corresponding server methods. With this set-up the server is ready for incoming client requests.

6.2.3. The util package

The server's `util` package contains the class `ServerUtil` having one utility method:

- `Document addUUIDsToDocument(Document document);`

This method accepts a document object of the type `org.w3c.dom.Document` and adds UUIDs to each element node in the document. In order to do this, the document is normalized (removing unnecessary text nodes) and a UUID is created for each element node using the same method as in the `newUUID(...)` method of the `CEFXUtil` class (see chapter 6.1.6).

6.3. Computer Networking issues

6.3.1. Network bandwidth and delay

With the constant development in computer networking technologies over the past few years, network bandwidth has increased dramatically. While a few years ago, in the early 1990s, a 10 MBit/s local area network (LAN) was state of the art, today 1 GBit/s or 10 GBit/s LANs are standard at home or in companies and organisations. With the commercial application of the digital subscriber line (DSL) technology the internet connection speed also increased dramatically and today the standard internet connection data rate via DSL is between 1 MBit/s (ADSL) and 50MBit/s (VDSL)[63]. This enormous increase in bandwidth facilitated new internet applications such as internet telephony (VoIP), video telephony, IPTV and on-line gaming.

Thus, when developing a collaborative real-time editing application, the network bandwidth issue is usually not as big as it used to be a few years ago. At least not when developing a real life practical application that can rely on an organisation's 10 GBit/s LAN. However, when designing a collaborative real-time editing application, it still is important to keep the bandwidth in mind and not to squander it - especially

[63] Provided by Deutsche Telekom: T-Home Entertain Comfort Plus VDSL, T-VDSL2 50 in selected regions in Germany, Source: http://de.wikipedia.org/wiki/VDSL, retrieved October 30, 2007

when the application is required to work over the internet.

Another networking issue is the network delay. The network delay in an IP network is generally the round trip delay for an IP packet. IP network delay comprises the sum of transmission delays and queuing delays experienced by a packet travelling through the collection of routers, switches and other hardware that comprise the network[64]. This is particularly an issue when transmitting information over the internet. IP network delays can range from just a few milliseconds to several hundred milliseconds.

When designing CEFX, the networking delay and network bandwidth issues were taken into consideration. Network delay was an important issue in the design of the CMAX concurrency control algorithm. The network delay can cause operations to arrive "out-of-order" at a client's site and the concurrency control algorithm must be able to cope with this. Thus CMAX orders the operations independently of their arrival time at a client's site, using the total ordering relation (see chapter 3 and 4). This allows to synchronise a shared document independently of any networking delay, even if the delay is very long.

Real-time collaborative systems that work on the basis of transmitting low-level input/output data such as key-strokes, mouse movements and display pixel data (see chapter 5.2) encounter the problem of high network traffic. The high network traffic is caused by the transmission of every single key stroke or mouse movement and display pixel data, generating many IP packets containing only little information. A field study at Airbus in 2007 showed that the used collaboration system (HP Remote Graphics) required about 4 MBit/s of bandwidth (synchronous) when 5 users work in parallel in order to achieve an acceptable performance (Guenther and Hörich 2007).

In order to synchronise a shared XML document, CEFX transmits `Operation` objects containing XML nodes in a serialised form over the network. Operations are transmitted much less frequently than mouse movements or key strokes, producing less network traffic. As the `Operation` objects do not carry very much information – basically they contain the XML node and the state vector – they are not very large, which saves bandwidth. Only at the beginning of an editing session is the complete

---

64 Source: http://en.wikipedia.org/wiki/Network_delay, retrieved October 30, 2007

XML document transmitted over the network, when a client connects. Thereafter the network transmission is reduced to the `Operation` objects. That is, if the awareness widgets do not require to transmit mouse movements or key strokes. Network traffic was one reason for the decision to transmit only mouse selection events when designing the default CEFX awareness widget. However, the plug-in architecture of CEFX allows easy configuration of the awareness mechanisms or deactivation of them, if the available network bandwidth requires it.

In order to optimise the transmission of the `Operation` objects, they could be compressed before transmission and decompressed when they arrive at the target site. This would additionally reduce the network traffic and may be a task for the future development of CEFX. Compressing `Operation` objects could be of interest, if CEFX is used in a mobile environment, for example on mobile phones, where network bandwidth is still an issue.

6.3.2. Networking software issues

6.3.2.1. Remote Method Invocation

CEFX uses the Java Remote Method Invocation (RMI) API in order to transmit information over the network. RMI is an often used, very efficient and reliable way for remote procedure calls. As it is part of the standard Java runtime, it is well documented and easy to use. These were the main reasons for choosing RMI when designing CEFX. However, the standard Java RMI implementation has some drawbacks in terms of connectivity. It is required that each network node (computer) that is to be connected using RMI must be in the same (virtual) network. This means that each computer must be able to "reach" the other ones directly by using the TCP/IP protocol.

This is usually not the case when a computer is connected to the internet through an internet service provider via a dial-up or a DSL connection. Depending on the network topology, a number of routers (NATs) and firewalls could lie between the network nodes preventing the computers from directly connecting. One solution to the problem is a Virtual Private Network (VPN) (Gleeson et al. 2000). Another alternative is using a so called Peer-To-Peer (P2P) technology. The following sections

briefly discuss these two approaches.

6.3.2.2. Virtual Private Networks

A VPN is often used by companies or organizations and allows confidential communication over a public network such as the internet. The network traffic is encrypted and is carried on top of standard protocols such as TCP/IP. The problem with a VPN is that it is relatively complicated to set-up and requires the installation of special VPN software and hardware components. Although encryption ensures a secure internet connection it also can slow down the network throughput. The advantage is, that with a VPN, RMI could be used for connecting clients over the internet in spite of the network topology.

6.3.2.3. JXTA

JXTA (Juxtapose) is an open source Peer-To-Peer (P2P) platform created by Sun Microsystems in 2001[65]. It defines as a set of XML based protocols that allow any device connected to a network to exchange messages and collaborate in spite of the network topology. JXTA was designed to allow a wide range of devices - PCs, mainframes, cell phones, PDAs – to communicate in a decentralized manner[66]. The devices that are connected using JXTA are called "peers". Peers create a virtual overlay network allowing direct interaction even when some of the peers are behind firewalls and routers (NATs). A Java implementation of JXTA is available and could be used to integrate it into CEFX by extending the `NetworkController` component. This may also be a task for the future CEFX development.

6.4. Supporting awareness mechanisms

The CEFX prototype system developed in this work only supports very simple awareness mechanisms. It notifies a user of the mouse selections and XML document changes of other users in the form of text messages. This may be not sufficient for a productive system but enough for a proof of concept prototype. Providing complex awareness widgets goes beyond the scope of this dissertation.

---

[65] Official JXTA website: http://www.jxta.org/ , retrieved October 30, 2007

[66] Source: http://en.wikipedia.org/wiki/JXTA , retrieved October 30, 2007

However, as mentioned before, the plug-in architecture of CEFX allows third party developers to easily integrate their own awareness mechanisms in the form of awareness widgets. This is done in two steps. The first step is to develop a widget plug-in that implements the `AwarenessWidget` interface. The easiest way to do this is to derive a new class from the `AbstractAwarenessWidget` class. This will leave the following methods to be implemented for the developer:

- `void init();`
- `boolean hasInterestIn(AwarenessEvent event);`
- `void notifiyOfAwarenessEvent(AwarenessEvent event);`

The method `init()` is called upon initialisation of the widget class. In this method the widget should create the user interface which will contain the information that is to be presented to the user. The method `hasInterestIn(...)` is called just before the widget is notified of an event and should contain code that checks if the given event is relevant to the widget. For example a widget that will notify the user of key strokes may not be interested in mouse events or vice versa. The `notifyOfAwarenessEvent(...)` method is called by the framework when the corresponding event (e.g. key event) has occurred and should contain the code that presents the event in some way to the user.

The second step is to use the extension point mechanism of CEFX to inform the framework of the new plug-in by registering it in the CEFX configuration file (`cefx.xml`). The framework will then load the new awareness widget at start-up. Registering a plug-in is done by adding the following code to the `cefx.xml` configuration file:

```
<ExtensionPoint id="AwarenessExtensionPoint" name="Awareness Extension">
<AwarenessExtension id="AwarenessExtension">
      <AwarenessWidget class="my.own.AwarenessWidget">
            <AwarenessEvent>KEY_EVENT</AwarenessEvent>
      </AwarenessWidget>
//...
</ExtensionPoint>
```

After registering the widget, it has to be activated by adding the following code to the `cefx.xml` configuration.

```
<Activation>
      <ExtensionPointRef>
            AwarenessExtensionPoint
      </ExtensionPointRef>
</Activation>
```

The text between the `ExtensionPointRef` tags has to be identical with value of the `ExtensionPoint's` attribute `id`.

The awareness widget integration mechanisms could also be used to integrate other kinds of collaboration supportive widgets. For example it could be used to integrate a simple chat widget that allows the users to communicate directly, typing text messages in order to coordinate their work more easily.

The development of new more supportive awareness widgets is a task for the future development of CEFX.

6.5. Conclusions

The CEFX framework has been successfully developed and will be applied in the prototype collaborative editing system described in chapter 7. It consists of a server and a client part which are structured in different Java packages depending on their functionality.

Although other technologies such as peer-to-peer (JXTA) could have been used for this work in order to connect the distributed clients, the Java Remote Method Invocation (RMI) technology was selected because it was found to be simpler to apply yet adequate for the purpose of this thesis research. After testing and refinement of the CEFX components, the framework was successfully integrated into a SVG graphics editing application for a proof of concept prototype which is discussed in chapter 7.

Chapter 7. Integration of CEFX into an existing single-user application

As mentioned in chapter 5.3 the transparent extension of an existing single-user application with CEFX is achieved by connecting it with and integrating the CEFX DOM Adapter (DA). The DA integration can be achieved in different ways.

One way is to directly connect the DA to the single-user application's internal data model. Another way is to connect CEFX with the application through a DOM/DOM translation layer that forwards the application's data model events to the DA and vice versa. The last possibility is to develop a DOM/API translation layer which translates application and runtime (OS) events into DOM events and vice versa.

For the proof of concept prototype implementation, the first possibility was chosen – directly connecting the DA with the application's data model. The objectives were, not to change the extended application's source code and to integrate CEFX with minimal effort. This was achieved by using a new programming paradigm: Aspect Oriented Programming (AOP).

The GLIPS Graffiti editor[67] was selected as single-user application for the transparent integration of CEFX. The GLIPS Graffiti editor is an open-source cross-platform SVG graphics editor developed by ITRIS[68]. It enables the creation of regular SVG files. As GLIPS is a Java application, it was necessary to use an aspect-oriented extension to the Java programming language. In this case AspectJ was used for the development of the required aspects.

The following sections explain the basic concepts of AOP briefly and how it was used to integrate CEFX into the GLIPS editing application in order to enhance it with collaborative real-time editing functionality.

---

[67] GLIPS Graffiti SVG Graphics Editor, http://glipssvgeditor.sourceforge.net/ , retrieved October 30, 2007

[68] For more information on ITRIS see: http://www.itris.fr/ , retrieved October 30, 2007

7.1. Extending GLIPS

7.1.1. Aspect Oriented Programming

Aspect-oriented programming (AOP) or Aspect-oriented Software Development (AOSD) is a programming paradigm for the separation and encapsulation of concerns, especially cross-cutting concerns, within a software. A concern within an software is, for example, logging. Logging, in the terminology of AOP, is an example for a cross-cutting concern because, within the source code of a software, logging code usually exists at many different locations. One advantage of AOP in comparison to, for example, object oriented programming is that these cross-cutting concerns can be located and managed at a single source code location, reducing maintenance effort and and obtaining more clarity. The most popular AOP language is AspectJ[69], developed by Gregor Kiczales et al. at Xerox PARC[70].

In order to encapsulate cross-cutting concerns at one place, so called aspects are defined which are then integrated into the software not earlier than at compile time. An aspect can alter the behaviour of the base code (the non-aspect part of a program) by applying advices (additional behaviour) at various join points (points in a program) specified in a quantification or query called a pointcut (that detects whether a given join point matches). An aspect can also make binary-compatible structural changes to other classes, like adding members or parents[71].

7.1.2. AOP integration of CEFX

The CEFX DOM Adapter (DA) is the entry point to the framework. At the beginning of each collaborative session, when a document is opened by the user, the DA is provided with a reference to the Document Object Model (DOM) of the application. This is done by one of the advices that are called if a certain join point within the application is executed. For each local modification of the DOM, the DA creates an operation which is then executed and propagated to the other sites. Incoming opera-

---

69 The AspectJ Project, http://www.aspectj.org , retrieved October 30, 2007

70 PARC, Palo Alto Research Center, Inc., http://www.parc.xerox.com/ , retrieved October 30, 2007

71 Aspect-oriented programming, http://en.wikipedia.org/wiki/Aspect-oriented_programming , retrieved October 30, 2007

tions are – after passing different synchronisation steps – eventually executed directly on the DOM of the application, as if they were executed locally by a user action.

When extending the GLIPS Graffiti editor, the cross-cutting concerns that were interesting were the modification of the applications data model. The first step was to identify the relevant join points that are executed when the data model is modified. User operations such as drawing a line, changing the colour of an object or deleting an object, modify the applications data model.

The GLIPS Graffiti editor is used to create and edit Scalable Vector Graphics (SVG). As SVG is an XML document format, GLIPS uses an XML document internally as data model. For the modification of the XML document, GLIPS makes use of the DOM API. For rendering the SVG graphics to the screen, the Apache Batik library is used[72]. Batik provides its own implementation of the DOM which complies to the W3C DOM specification.

This simplified the identification of the relevant join points. All calls to functions defined by the W3C DOM API were possible candidates for a relevant join point.

The next step was to write an Aspect class that encapsulates the cross-cutting concerns at one place. The Aspect class is similar to a Java class and can contain normal Java code and, additionally, AspectJ components. A simple example:

```
public aspect DOMAccessAspect {
…
}
```

The defined Aspect is then woven into the application's code at compile time. The application's source code is not modified. The Aspect class defines point cuts that are executed by AspectJ when the matching join points are reached within the application.

The following calls to the W3C DOM API methods were identified as most relevant to the modification of the GLIPS data model:

- `Node appendChild(Node newChild);`
- `Node insertBefore(Node newChild, Node refChild);`

[72] Batik SVG Toolkit, http://xmlgraphics.apache.org/batik/ , retrieved October 30, 2007

- `Node removeChild(Node oldChild);`
- `void setAttributeNS(String namespaceURI, String qualifiedName, String value);`
- `void setAttribute(String name, String value);`

The first three methods are defined by the `org.w3c.dom.Node` interface, the last two by the `org.w3c.dom.Element` interface. The methods `appendChild(...)` and `insertBefore(...)` are called whenever a node is added to the document. This is the case, for example, if the user draws a line in the SVG document. The method `removeChild(...)` is called whenever a node is removed from the document. That is the case if the user deletes an object from the document. The methods `setAttributeNS(...)` and `setAttribute(...)` are called if, for example, the user changes the colour of an object. In SVG the colour information of an object is contained in the value of the `style` attribute.

Figure 7.1 shows a scenario of what happens if, for example, the colour of an object is changed.

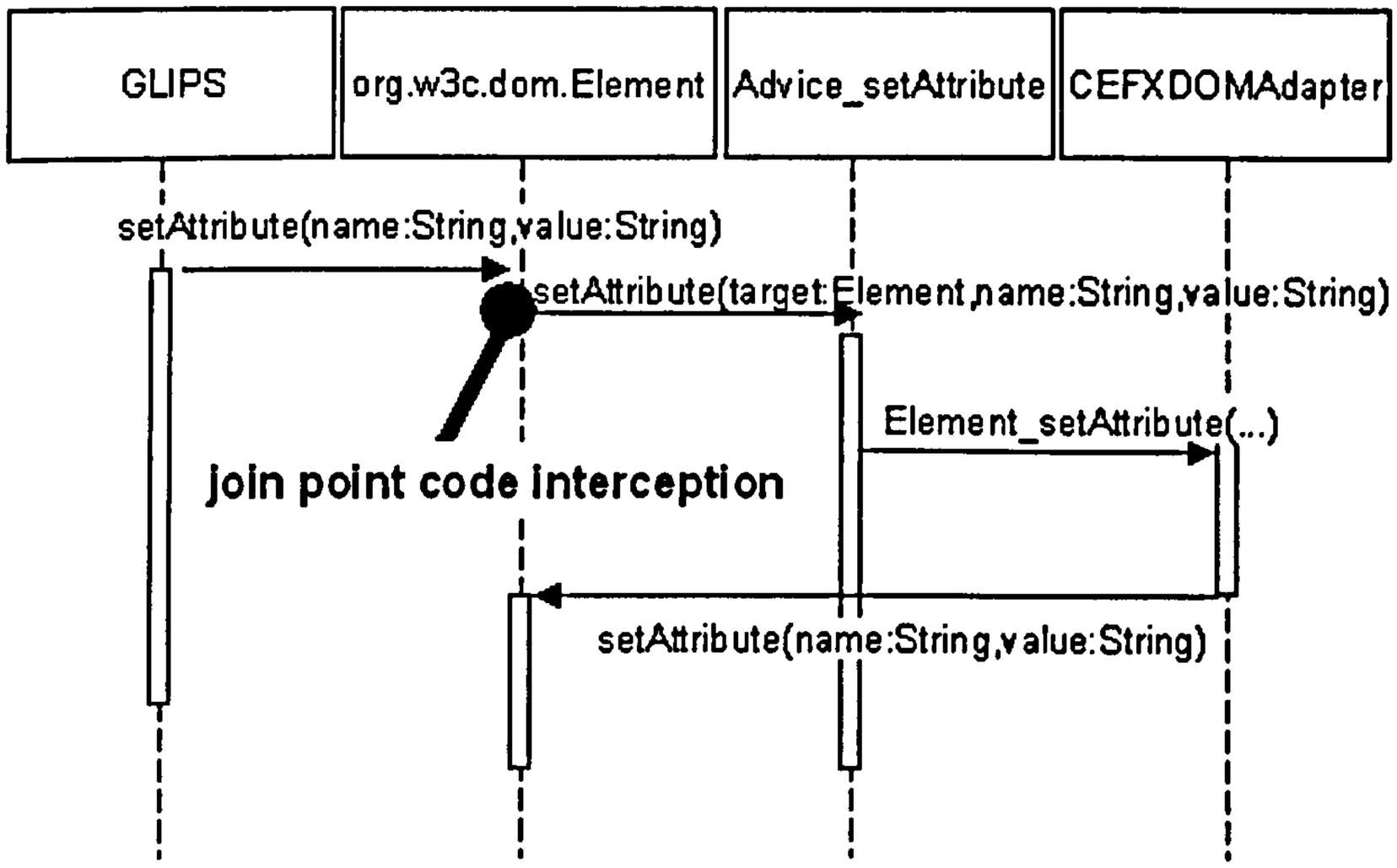

Figure 7.1: Scenario of code interception

Whenever the method `setAttribute(...)` on an element node of the XML document is called, the call is intercepted by AspectJ. Instead of directly executing the

code of the DOM implementation provided by Batik, the defined pointcut of the aspect class selects the relevant join point and the specific advice is applied. The advice then delegates the call to the DA. The DA creates an update operation which is handled by the `CEFXController` and asynchronously propagated to all sites of the current editing session by the `NetworkController`. The operation is instantly executed on the local element node and the attribute value is updated.

Pointcuts pick out interesting join points in the execution of a program. These join points can be, for example, method invocations and executions. An AspectJ pointcut definition gives a name to a pointcut. The code snippet below shows how the pointcut of the above scenario is defined in AspectJ.

```
...
pointcut setAttributePC(Element p, String attr, String value):
target(p) && args(attr,value)
&& call( void setAttribute(String,String))
&& !within(DOMAccessAspect);
...
```

This pointcut picks out all join points matching a call to a method with the same signature and parameters as given in the `call(...)` statement. For each pointcut, an advice is defined, that contains the code that is to be executed if the pointcut is met. For each of the relevant pointcuts an advice is defined. The advice, that is executed for the above pointcut is shown in the following code snippet.

```
...
void around(Element p, String attr, String value):
setAttributePC(p,attr,value)
{
Advices.setAttribute(p,attr,value);
}
...
```

The static method `setAttribute(...)` of the class `Advices` is called here. As discussed in chapter 6.1.6, the class `Advices` is a helper class containing the advice's code. This was done for clarity reasons and in order to separate the Java code from

AspectJ code. The code that is executed here is shown in the following code snippet.

```
...
public static void setAttribute
(Element p,String attr,
 String value)
{
 CEFXDOMAdapter doma = CEFXUtil.getDOMAdapter();
 if (doma != null) {
  if(doma.isCollaborationReady()) {
   doma.Element_setAttribute(p,attr,      value);
       return;
  }
 }
 p.setAttribute(attr, value);
}
...
```

First a reference to the `CEFXDOMAdapter` is retrieved by calling the static method `getDOMAdapter()` of the `CEFXUtil` class (see chapter 6.1.6). If the DA is already initialised and ready for collaboration, the call to `setAttribute(...)` is delegated to the corresponding DA method. If the DA was not properly initialised, the `setAttribute(...)` of the target object is called directly, setting the new value to the attribute. This is done for example, if the client is not connected to a collaboration session.

For all other relevant pointcuts the same procedure was applied. The following code shows an example on how the other advices were defined using anonymous pointcuts.

```
...
Node around(Element p, Element c): target(p) && args(c) &&
call(Node appendChild(Node)) && !within(DOMAccessAspect)
{
return Advices.appendChild(p,c);
}
...
```

Additionally to the advices for the manipulation of the data model, advices for creating or loading of a document and the initialisation of the render context are needed. Creating a new SVG document or loading and parsing of an existing document is handled by the `SAXSVGDocumentFactory` class provided by Batik. Rendering of an SVG document to the screen is handled by the `SVGCanvas` class (the render context) provided by GLIPS. When a document is opened, the DA is provided with a reference to it, in order to integrate changes from the remote sites. After the execution of a remote operation, the render context of the application is notified in order to repaint the document. For this reason, the DA is provided with a reference to the application's render context. The following advice code is executed, when a document is opened by the user.

```
...
SVGDocument around(SAXSVGDocumentFactory fac, String uri):
target(fac) && args(uri) && call(SVGDocument
createSVGDocument(String)) && !within(DOMAccessAspect)
{
//Initalisation of the DA
 CEFXDOMAdapter da = new CEFXDOMAdapterImpl();
//Proving DA with factory reference
 da.setDocumentFactory(fac);
//Creating the document and
//providing DA with a reference to it
SVGDocument doc = (SVGDocument) da.createDocument(uri);
...
//Exception handling
...
}
...
```

The DA is also provided with a reference to the document factory. As discussed in chapter 6.1.1.1, this is required for example, if a session for the opened document already exists. In this case, CEFX loads the document from the server and handles the document parsing and initialisation.

The advice for setting the render context is called when the `SVGCanvas` is initialised

within the application. The following code illustrates how this is achieved.

```
...
after(SVGCanvas panel): target(panel) && call(* initializeCanvas(*))
{
 CEFXDOMAdapter da = CEFXUtil.getDOMAdapter();
 if (da != null) {
  da.setRenderContext(panel);
 }
}
...
```

The `SVGCanvas` class is derived from `javax.swing.JlayeredPane` and provides a method `initializeCanvas()`. The advice is executed after the initialisation method has been called. Thus in this case the method call is not intercepted. The advice is merely used for notifying CEFX of the initialisation and providing it with a reference to the render context.

7.1.3. Integrating awareness support

The discussed advices were used to integrate CEFX in a way that satisfies the requirements of communication, session management and concurrency control. In order to satisfy the requirement of group awareness, additional effort is necessary.
As discussed in chapter 6.1.7, CEFX provides a simple awareness widget that allows notification of each user in a collaborative session; for example, on other user's mouse selections. This can help a user to get an understanding of what other users are working at. The awareness widget is a little window, controlled by the CEFX client and is independent of the extended application. The application is not aware of that widget.

For the integration of the awareness support provided by CEFX into the GLIPS editor, additional advices can be used. Java applications make use of certain interfaces from the `java.awt.event` package in order to retrieve information on mouse clicks and mouse movements. In the following example it is shown how CEFX is notified of a user's mouse clicks.
A new Aspect class containing advices for mouse events was developed:

```
public aspect SelectionAspect {
…
 after(MouseEvent event):
 args(event) && execution(
 void mousePressed(MouseEvent))&&
 !within(SelectionAspect)
 {
  EventPropagator.propagateEvent(event,
    AwarenessEventTypes.MOUSE_EVENT.toString(),
    AwarenessEventDescriptions.MOUSE_PRESSED.toString(),
    EventPropagator.SCOPE_EXTERNAL, null);
 }…
}
```

The above code snippet shows the advice that is executed when the user presses the mouse button. The static method `propagateEvent(...)` of the `EventPropagator` class is called in this advice. The `EventPropagator` then creates a new `AwarenessEvent` object and forwards the event to the `AwarenessController` which in turn propagates the event to the other clients in the session (see chapter 6.1.7.3).

The same kind of advices can be implemented for all other kinds of user events such as typing the keyboard or mouse movements. Using AOP here allows a transparent integration of awareness mechanisms into the application.

7.2. Integration Summary

To summarize, using aspect-oriented programming for the integration of CEFX into the GLIPS Graffiti editor did not require much programming effort. Only seven advices were needed to provide GLIPS with the basic collaboration functionality. Five of the used advices were related to the DOM API and can be reused for other applications using the DOM. One advice was specific to the Batik library and one was specific to the application. The overall performance of the application did not change noticeably.

AspectJ is one of many existing implementations of AOP. In this case, for example, HyperJ[73] could alternatively be used.

---

[73] HyperJ Overview (Tarr, P). http://www.alphaworks.ibm.com/tech/hyperj , retrieved October 30, 2007

However, one requirement in order to integrate CEFX using AOP is that the target application is written in a language that is supported by AOP.

AOP implementations exist for many different languages and platforms such as Java, C#, VB.NET, JavaScript, C/C++, Lua, Python, Ruby, Perl, PHP, Common Lisp and many others.

Some AOP implementations require recompilation of the application's source code in order to weave the generated aspect code into it. Other AOP implementations do not require source code. AspectJ for example, supports byte-code weaving and advanced load-time weaving. This allows using AOP without access to the application's source code, which makes it suitable for the extension of commercial applications.

The integration of CEFX into the GLIPS application had the advantage that GLIPS uses the DOM for accessing its data model. This simplified the identification of relevant join points. For applications that do not use a standard interface for modifying their data model, the identification of the join points may be more difficult, but still feasible.

It can be assumed that using standardised data model interfaces and aspect-oriented concepts can dramatically reduce implementation efforts in comparison to other approaches using window event translation and application specific programming interfaces. This work shows how little the effort is to transparently extend a single-user SVG editing application using this approach.

7.3. Installation and set-up of the CEFX proof of concept prototype software

The CEFX proof of concept prototype consists of the GLIPS Graffiti SVG editing software and the CEFX client and server classes. All relevant Java classes and JAR archives are contained in the `CEFXDemo` folder on the CD that accompanies this thesis. The folder contains the Java source code and the binary code that is executed. The CEFX API documentation can be found in the folder `CEFXDemo\docs`. The GLIPS Graffiti software's source code can be found in the folder `CEFXDemo\GLIPSGraffiti`. The CEFX server source code is located in `CEFXDemo\CEFXServer` directory. The binary code is located in the

CEFXDemo\serverRuntime folder. The CEFX client source code is located in the CEFXDemo\CEFX folder. The CEFX client binary code is located in the client runtime folder. Two folders exist, one for each client. These are CEFXDemo\client1Runtime for the first client and CEFXDemo\client2Runtime for the second client. The two client runtime folders allow two clients to start on one computer for testing purposes.

The client runtime folders also contain all necessary GLIPS Graffiti binaries and the AspectJ runtime classes.

In order to install the CEFX demonstration on a computer, the CEFXDemo folder is copied to the local hard drive first. The next step is to set-up the demo.

7.3.1. Setting up the CEFX demonstration

The CEFXDemo folder contains a file called startCEFXDemo.bat. The following lines of this file have to be edited.

```
#Path to the installation directory of the CEFX Demo

set CEFX_LOCATION=D:\\CEFXdemo\\

#Path to the Java Development Kit

set JAVAPATH=C:\Programme\Java\jdk1.5.0_07\bin
```

The line with the CEFX_LOCATION variable has to point to the installation directory on the local hard drive. If for example the CEFXDemo is located at C:\Programs\CEFX-Demo, then this line has to be changed to:

```
set CEFX_LOCATION=C:\\Programs\\CEFXDemo\\
```

The line with the JAVAPATH variable has to point to the location of the JDK. The Sun JDK 1.5 has to be installed on the target machine. It is not sufficient to install the Sun Java Runtime Environment (JRE) as it does not include the RMI registry software.

The next step is to set-up the CEFX network properties for the clients. The file named network.properties in each client runtime folder contains the following (or similar) lines:

```
NetworkControllerImpl.server.hostname=192.168.0.20
NetworkControllerImpl.server.port=2000
NetworkControllerImpl.server.connection.name=CEFXServer
NetworkControllerImpl.client.hostname=192.168.0.21
NetworkControllerImpl.client.port=3000
NetworkControllerImpl.client.connection.name=CEFXClient
NetworkControllerImpl.client.id=1
NetworkControllerImpl.client.name=Client1
```

The first line specifies the IP address or hostname of the server. The address in this line has to be changed to the IP address of the computer that runs the server. For example, if the computer that runs the server has the IP address `10.21.0.25`, then this line has to be changed to:

```
NetworkControllerImpl.server.hostname=10.21.0.25
```

The next line specifies the server port. This line should not be changed unless the port is already used by another application running on the same machine. The property `NetworkControllerImpl.server.connection.name` defines the name that is used to register the server with the RMI registry. This should not be changed. Next, the `NetworkControllerImpl.client.hostname` property has to be set to the IP address or hostname of the machine that runs the client. The above values show the settings of the first of the two clients. If the clients and the server run on the same machine, this can be set to `localhost`.

The properties for the client port and connection name should also not be changed. The last two properties specify the client id and the client name. In this case the first client has the properties set to `id=1` and `name=Client1`. This should be left as it is. In the `network.properties` file of the second client, these properties are set to:

```
NetworkControllerImpl.client.id=2
NetworkControllerImpl.client.name=Client2
```

If a third client exists, these properties have to be set accordingly. For example for the third client the `network.properties` file would contain the following.

```
NetworkControllerImpl.server.hostname=192.168.0.20
```

```
NetworkControllerImpl.server.port=2000

NetworkControllerImpl.server.connection.name=CEFXServer

NetworkControllerImpl.client.hostname=192.168.0.23

NetworkControllerImpl.client.port=3030

NetworkControllerImpl.client.connection.name=CEFXClient

NetworkControllerImpl.client.id=3

NetworkControllerImpl.client.name=Client3
```

7.3.2. The document repositories

Each client and the server have a temporary repository folder where the currently edited document is stored. The server's document repository is located in the `CEFXDemo\ServerTempRepository` folder. The client's repository has to be specified in the `client.properties` file, which is located in each client's runtime directory. This property file contains the following entries:

```
CEFXControllerImpl.Tempfile.Location.URI=client1Runtime\\Client-
TempRepository

CEFXDOMAdapterImpl.Repository.Name=CEFXRepository
```

The first property specifies the client's temporary repository which is used for storing the currently edited document. The value is a relative path to the repository folder starting from the CEFX installation folder. For example, if the CEFX demonstration software is located in `C:\Programs\CEFXDemo`, the first client's repository is located in `C:\Programs\CEFXDemo\client1Runtime\ClientTempRepository`. This property has to be set accordingly for each client. In the case of the second client, the value should be set to `client2Runtime\\ClientTempRepository`.

The second property specifies the relative path to the `CEFXRepository` folder starting from the installation folder. For example, if the software is installed in `C:\Programs\CEFXDemo` the repository folder is located in `C:\Programs\CEFXDemo\CEFXRepository`. This property should not be changed. The `CEFXRepository` folder contains all documents that can be edited collaboratively. CEFX checks the path of each file that is opened by the user and if it is located

in the `CEFXRepository`, a new collaborative editing session is initialised. If the document is not within the `CEFXRepository`, it will be ignored by CEFX and can only be edited in single-user mode.

7.3.3. Starting an editing session

After configuring CEFX, the demonstration software is started by double clicking on the `startCEFXDemo.bat` file (this batch file will only work on the Windows operating system). This will start the CEFX server and two instances of the GLIPS Graffiti SVG editing application. Figure 7.2 shows a screen shot of the GLIPS Graffiti application after starting it.

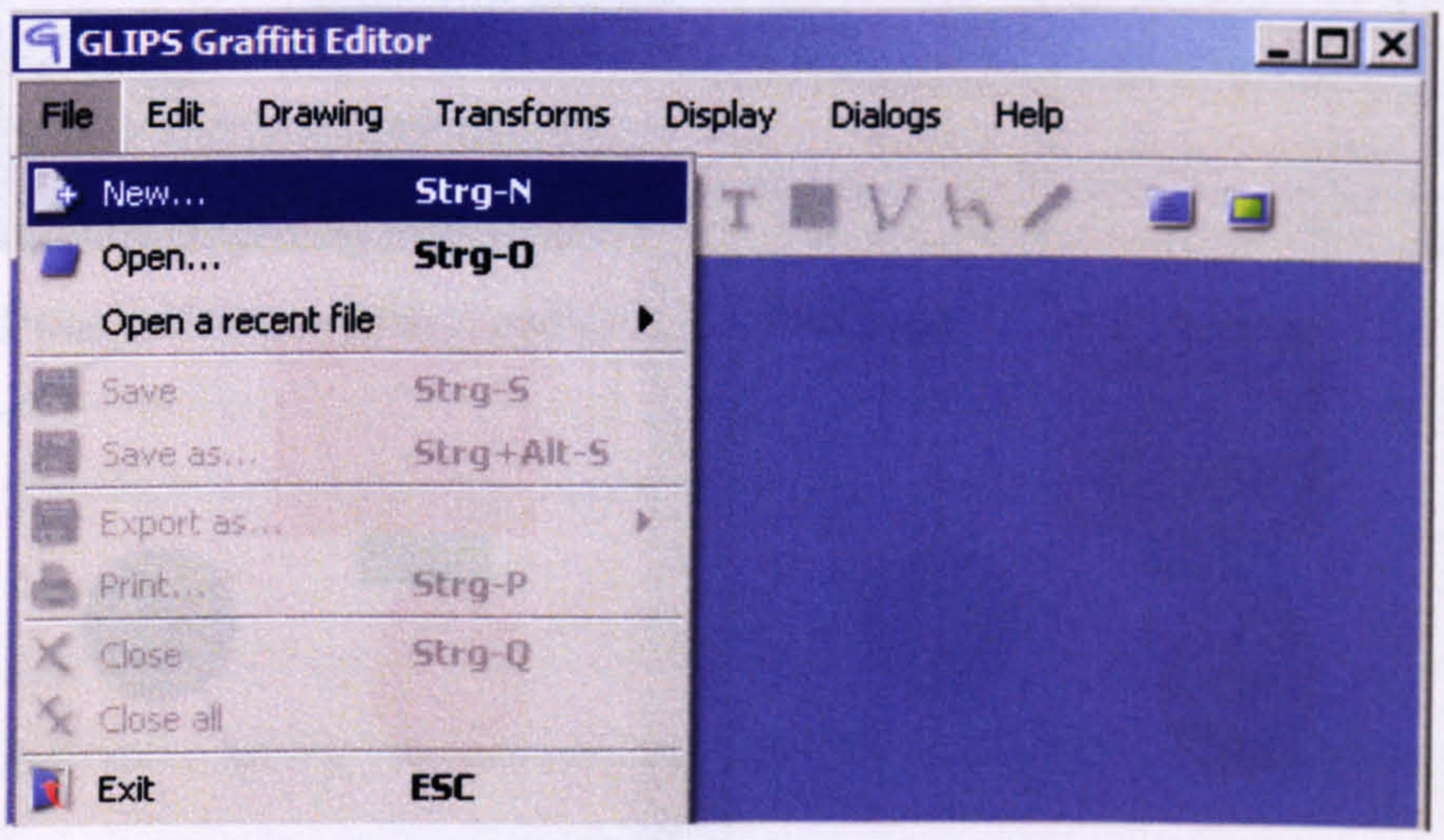

Figure 7.2: The GLIPS Graffiti editor user interface

In order to start a new collaborative editing session, a new document has first to be created and stored in the `CEFXRepository` folder. After storing the document, it has to be opened (using the `Open...` menu item as shown in figure 7.2) from the `CEFXRepository` folder. After opening the document, CEFX will open a new collaborative real-time editing session for the document. The next step is to open the same file using the second GLIPS Graffiti instance. After the document was loaded in both clients, it can be edited collaboratively and is stored periodically by the server in its repository. The next time the document is opened again, it will be loaded from the server and not from the client's repository.

When the editing session starts, CEFX initialises the awareness widgets. Figure 7.3 shows both clients and the awareness widgets. The awareness widget windows are independent of the editing application and can be closed or minimized if they are not used. CEFX can also be configured to start other awareness widgets or none as explained in chapter 6.4.

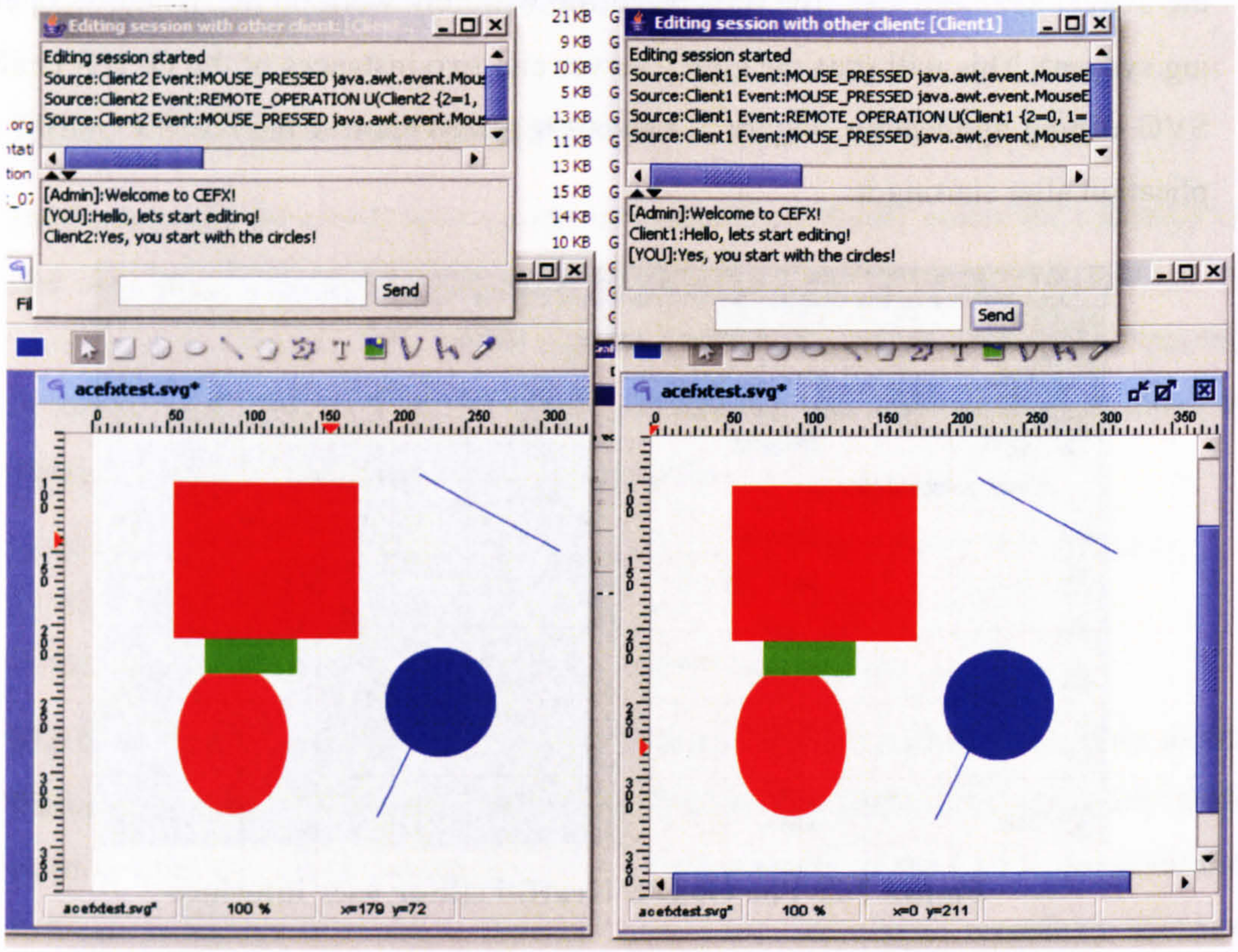

Figure 7.3: A collaborative editing session with GLIPS and CEFX

The CEFX demonstration software runs on one computer with two clients. In order to collaboratively work with two or more clients from different computers, the software has also to be installed and configured as described above on the other devices. In order to work on a shared document at each client site, an empty SVG document with the same name has to be created in the `CEFXRepository` folder. When opening the document at each site, CEFX will load the shared document from the server. The local document is only a dummy. It does not need to contain anything and is solely used to identify the document that should be retrieved from the server.

The CEFX proof of concept prototype software was tested solely on the Windows XP operating system. However, because it was developed in the Java programming language it should run on other operating systems (e.g. Linux) without modification, provided that an appropriate Java runtime is available for that system.

7.4. Conclusions

A new framework integration concept based on the Aspect-Oriented Programming (AOP) paradigm was implemented. This is the first time that AOP has been used to successfully integrate a collaborative editing framework into an existing editing application. This thesis argues that using AOP and concentrating on the application's data model instead of an application, operating system or GUI library API, should reduce the effort of implementation. This is because it simplifies the process when integrating a collaboration framework into an existing single-user application. The proof of concept prototype that was developed in this thesis, shows that the effort involved was within a practical and efficient time scale.

Chapter 8. Final discussion

8.1. Summary

Computer Supported Cooperative Work (CSCW) is an interdisciplinary field of research dealing with group work, cooperation and their supporting information and communication technologies. One part of CSCW is the so called Real-Time Collaborative Editing (RTCE). RTCE investigates the design of systems which will allow several persons to work at the same time (in real-time) on the same document without risking any inconsistencies.

Existing CSCW editing applications in contemporary research focus on one or only a small number of document types. However, this dissertation investigated the development of a software framework which will allow collaborative work on basically any XML document type in real-time, representing a more general solution to the problems of real-time collaborative editing.

In many areas XML is becoming the standard interchange and data format. More and more applications not only support XML as an exchange format but also use it as their data model or default file format.

This research study contributes a novel software framework concept allowing software engineers to develop new collaborative real-time XML editing applications without having to invest much time and effort into collaboration issues. The framework is based on a flexible plug-in architecture including support for concurrency control for XML documents, awareness mechanisms and optional locking of document parts. Additionally, this thesis contributes a novel framework integration strategy that allows transparent extension of existing single-user editing applications with real-time collaborative editing functionality.

The first chapter of this thesis discussed the XML document structure, the well-known problems of concurrent editing on linear and hierarchical data structures in general (divergence, causality violation and intention violation) and existing concurrency control techniques.

After the introductory chapter 1, chapters 2 to 4 described the development and implementation of a consistency maintenance algorithm for XML documents (CMAX).

Chapters 5 and 6 dealt with the design and implementation of the Collaborative Editing Framework for XML (CEFX). The last part of the thesis (chapter 7) discussed the novel framework integration strategy based on the Document Object Model (DOM) and Aspect-Oriented Programming (AOP).

As proof of the above concept, an existing single-user SVG graphics editing application was transparently (without changing the source code of the application) extended using CEFX and AOP.

8.2. Conclusions

The first chapter explained technical aspects of XML and limitations of using hierarchical documents in terms of concurrency control. It identified the problems relevant to collaborative editing of XML documents.

Chapter two analysed the conflict probability when editing hierarchical data structures such as XML documents. The structure of XML documents was examined and a theoretical model for the probability of conflicts was developed. In order to find a formula for the conflict probability when editing hierarchical documents, equations for calculating the conflict probability when editing data structures such as linear documents and binary trees were developed. An equation for arbitrary XML document structures was then formulated. The next step was to conduct a Monte Carlo simulation for a variable number of users working on randomly generated XML documents of different sizes and structures. The purpose of the Monte Carlo simulation was firstly to demonstrate the devised equations for the simple binary tree structures were correct. Secondly to simulate concurrent editing on more complex structures in order to gain knowledge on the conflict probability in real concurrent editing situations. For this task an algorithm was developed that simulated a number of users concurrently editing XML documents of different structures and sizes. To get a more realistic simulation result, the XML documents used in the simulation were generated based on the properties found in the previous document analysis. The results of section 2.3.2 showed how the number of conflicts relate to the increasing size of documents and the number of workers on it. The same simulation was then conducted for a specific set of XML documents with a binary tree data structure. The results confirmed the calculated results for binary tree structures (figure 2.13). The simula-

tion results for arbitrary document structures did not confirm the equation derived for arbitrary XML document structures and it was not further investigated. In order to gain a better understanding of the behaviour of conflicts in a collaborative XML editing environment a simulation algorithm was developed for visualising a dynamic editing scenario (figure 2.14). The results of this analysis and the static simulation results were used as a catalyst for the development of a concurrency control algorithm for XML documents.

Chapter three discussed contemporary research projects on concurrency control algorithms for hierarchical structures and presented a new algorithm for the synchronisation of XML documents called "Consistency Maintenance Algorithm for XML" (CMAX). This chapter explained the properties of convergence, causality and intention preservation and how they are maintained using CMAX. The work described in this thesis is the first documented attempt to use universally unique identifiers (UUIDs) in a consistency maintenance algorithm to address nodes within an XML document and to use a new state vector swapping scheme (SVS) in order to preserve intentions. Using UUIDs has advantages over positional addressing schemes. Intentions can be preserved without the need to transform operations, reducing complexity and calculation time. In the case of resolvable conflicts the SVS scheme was used to solve the conflicts instead of applying operational transformation (OT), which is already used in other contemporary consistency maintenance algorithms. The consistency maintenance algorithm (CMAX) supports the synchronisation of any XML document type. As XML document types exist for many application areas, the CMAX algorithm can be generally used for the synchronisation of, for example, 2D and 3D graphic documents as well as text documents or spreadsheets.

The implementation of the CMAX algorithm in software was explained in chapter four. The software components that are responsible for the consistency maintenance are the `ConcurrencyController` and the `ConflictResolutionProvider`. The algorithms produced were tested using a specially developed simulation software. After refinement using the Java programming language, the CMAX algorithms were successfully integrated into the Collaborative Editing Framework for XML (CEFX).

Chapter 5 discussed contemporary collaboration systems and the CEFX software ar-

chitecture. CEFX is based on a logical hybrid architecture and uses a flexible plug-in concept where each component can be extended or replaced with a new implementation. This novel technique resulting from this research allows adaptation of the framework to specific application requirements with little effort. Whereas other research has used application specific software interfaces, this thesis pioneers a new method of using the Document Object Model (DOM) as a standard interface for the integration of collaboration functionality into an application, thus reducing the development effort dramatically compared with the approaches of others. CEFX provides a simple-to-use application programmer's interface (API) that enables the development of new collaborative XML editing applications or the extension of existing single-user applications. In addition to the consistency maintenance of XML documents, CEFX supports awareness mechanisms and optional locking of document nodes.

The implementation of CEFX and its components was discussed in chapter six. In order to connect the distributed clients, CEFX uses the Java Remote Method Invocation (RMI) technology. In connection with a virtual private network (VPN) CEFX clients can be securely connected over the internet. An alternative approach is the Peer-to-Peer (P2P) technology which was also briefly discussed in this chapter. Although technologies such as P2P could have been used for this work in order to connect the distributed clients, the Java Remote Method Invocation (RMI) technology was selected because it was found to be simpler to apply yet adequate for the purpose of this thesis research.

The CEFX framework has been successfully developed and was applied in the prototype collaborative editing system described in chapter 7.

In chapter 7 a new framework integration concept based on the Aspect-Oriented Programming (AOP) paradigm was implemented. This is the first time that AOP has been used to successfully integrate a collaborative editing framework into an existing editing application. This thesis argues that using AOP and concentrating on the application's data model instead of an application, operating system or GUI library API, should reduce the effort of implementation.

This is because it simplifies the process when integrating a collaboration framework into an existing single-user application. The proof of concept prototype that was developed in this thesis, shows that the effort involved was within a practical and efficient time scale.

8.2.1. Summary of achievements

A framework is developed that enables synchronous collaborative editing of XML documents, supports different awareness mechanisms and allows XML applications to be extended with the collaborative editing feature. Specialised editors for any kind of XML document type (e.g. SVG, X3D, DocBook) can use the framework to enable them to work collaboratively on documents. The framework is called Collaborative Editing Framework for XML (CEFX).

Outcomes which have contributed to this thesis are as follows:

- An analysis of XML document structures was conducted.

- An equation for the conflict probability for collaborative editing of binary trees was developed.

- Static and dynamic simulations of the conflict probability applied to collaborative XML editing were devised.

- A flexible consistency maintenance algorithm for XML documents supporting optional locking of document parts was developed.

- A flexible plug-in architecture enabling developers to extend CEFX with new awareness widgets, concurrency control mechanisms and conflict resolution schemes was developed.

- A new framework integration concept based on the Aspect-Oriented Programming (AOP) paradigm was implemented.

The collaborative editing framework for XML documents presented in this dissertation has a lot of potential for future developments. Real-time collaboration is only one possible application area. Another possible application area is, for example, the data integration of information systems such as Product Data Management (PDM) and Product Lifecyle Management (PLM) systems into authoring applications. A great challenge of the data integration is how to reduce the effort of connecting a proprietary authoring system with an information system. The aspect-oriented programming approach in combination with standard data formats based on XML – as described in this dissertation – could be a solution to this problem.

8.3. Future work

Collaborative editing systems is an area with many challenging issues that need to be resolved. The work presented in this thesis can be used as a foundation for the future research in XML based collaborative systems. Concerning the future development of CEFX, the following topics (that go beyond the scope of this thesis) have been identified for future investigation:

- A very important issue for the broad user-acceptance of a collaborative real-time editing systems is privacy support. An application should let the user decide, what of his activities others users are allowed to see. This can be achieved by awareness widgets that support masking of another user's activities. Only if the user is willing to show his activities can they be seen by others. In that case, the part of the document the user is working on must be locked by the user. This allows a certain amount of privacy that is often desired but does not exist in collaborative applications today. The desire for privacy also has an economical background. Consider an environment such as the automotive industry, where many different companies (subcontractors and suppliers) work together on a large document; for example an electrical circuit diagram for a new car. If the companies were using a collaborative editor which does not support privacy, everyone could see how a subcontractor works. That means others could retrieve knowledge of secret business information that is vital for the subcontractor to stay competitive. Future collaborative real-time editing systems that are to be used in a professional environment therefore, require some way of privacy sup-

port in order to become accepted by the users of such a system. CEFX supports optional locking of document nodes which could be used as the basis for privacy support. The next step would be to investigate how the changes that are performed by one user on a locked node can be hidden to the other users working on the same document.

- Group undo is an often required feature in a group editor, allowing a user to undo, not only his own operations, but also the operations of other users in an editing session. Group undo could be transparently applied in CEFX for example by providing a new widget that allows to view the operations in the history buffer and select one ore more of them for undo. The undo command in this case would generate a new operation which has the exactly inverted effect of the operation to be undone. This requires that operations are invertible as it is the case with CEFX. Applying such a functionality would not require any changes in the concurrency control algorithm of CEFX.
- In the CEFX proof of concept prototype an SVG graphics editing application was extended. In the future, more and other types of applications should be extended using CEFX. This will help to find limitations of the framework and to improve it.
- The current implementation of the CEFX server only supports one editing session at a time. This may be enough for a prototype implementation but makes it unsuitable for use in a professional environment. A future task will be to extend the server software so it will support many collaborative editing sessions at a time.
- The currently implemented default awareness widget notifies the user of an event by using text messages. This awareness widget could be improved with a more meaningful visualisation of the awareness event. An additional future task will be to develop other supportive awareness widgets.
- The documents that are edited using CEFX are stored at the server site in the local file system. A future task could be to integrate a version control systems in the server, which supports the retrieval of a version of a document at any time after an editing session ended.
- When storing a document locally at a client site from the editing application, the

UUIDs are still contained. They should be removed by the framework before storing. The documents in the `CEFXRepository` do not change when edited, only the files in the `ClientTempRepository` change. The client should write the current version of the documents to the `CEFXRepository` as well, when it disconnects.

References

Baecker, M. R., Nastos, D., Posner, I. R., Mawby, K.L. (1993). *The User-Centred Iterative Design of Collaborative Writing Software*. San Francisco, CA, USA. Morgan Kaufmann, pp. 775-782. ISBN 1-55860-246-1.

Begole, J.M.A. (1999). *Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems*. PhD Thesis. Virginia Polytechnic Institute and State University, Blacksburg.

Berners-Lee, T., Fielding, R., Frystyk, H. (1996). *Hypertext Transfer Protocol - HTTP/1.0. RFC 1945*. Retrieved July 15, 2007 from: http://www.ietf.org/rfc/rfc1945.txt.

Butterworth, P., Otis, A., Stein, J. (1991). *The GemStone Object Database Management System*. Communications of the ACM. Volume 34, Issue 10. ACM Publishing, pp. 64-77.

Chen, D. (2001). *Consistency Maintenance in Collaborative Graphics Editing Systems*. Brisbane. PhD Thesis. Griffith University.

Chen, D. (2001). *REDUCE - REal-time Distributed Unconstrained Collaborative Editing System*. Retrieved July 15, 2007 from: http://www.cit.gu.edu.au/~scz/projects/reduce/.

Davis, A., Sun, C., Lu, J. (2002). *Generalizing Operational Transformation to the Standard General Markup Language*. New Orleans, Lousiana, USA. ACM Press, pp. 58-67. ISBN 1-58113-560-2.

Davis, A.H., Sun, C., Lu, J. (2001). *Collaborative Editing of XML Documents – An Operational Transformation Approach*. Retrieved July 15, 2007 from: http://www.research.umbc.edu/~jcampbel/Group01/Davis_iwces3.pdf.

Dourish, P. (1997). *Extending Awareness Beyond Synchronous Collaboration*. Retrieved July 15, 2007 from: http://www.ics.uci.edu/~jpd/publications/chi97-awareness.html.

Ellis, C. A., Gibbs, S. J. (1989). *Concurrency Control in Groupware Systems*. Portland, Oregon, United States. ACM Press, pp. 399-407. ISBN 0163-5808.

Fidge, C. J. (1991). *Logical Time in Distributed Computing Systems*. Los Alamitos, CA, USA. IEEE Computer Society, pp. 28-33. Volume 24, Issue 8. ISBN 0018-9162.

Franaszek, P. A., Robinson, J. T., Thomasian, A. (1992). *Concurrency Control for High Contention Environments*. New York, NY, USA. ACM Press, pp. 304-345. Volume 17, Issue 2. ISBN 0362-5915.

Galli, R. (2000). *Data Consistency Methods for Collaborative 3D Editing*. Palma de Mallorca, Spain. PhD Thesis. Universitat de les Illes Balears.

Galli, R., Luo, Y. (2000). *MU3D: A Causal Consistency Protocol for a Collaborative VRML Editor*. Monterey, California, USA. ACM Press, pp. 53–62. ISBN 1-58113-211-5.

Gawlick, D. (1985). *Processing "Hot Spots" in High Performance Systems*. San Francisco, California, USA. Springer Verlag, pp. 249-251. ISBN 0-8186-0613-4.

Gleeson, B. et al. (2000). *A Framework for IP Based Virtual Private Networks - RFC 2764*. Retrieved July 15, 2007 from: http://www.ietf.org/rfc/rfc2764.txt.

Grudin, J. (1994). *Groupware and social dynamics: eight challenges for developers*. Communications of the ACM. Volume 37, Issue 1. ACM Press, pp. 92-105.

Guenther, J., Hörich, F. (2007). *Collaborative Engineering: Direct Data Integration, an option for the collaboration with suppliers*. Wolfsburg. Presentation at the ProStep iVIP Symposium 2007.

Gutwin, C., Roseman, M., Greenberg, S. (1996). *A Usability Study of Awareness Widgets in a shared Workspace Groupware System*. Boston, Massachusetts, USA. ACM Press, pp. 258-267. ISBN 0-89791-765-0.

He, F., Han, S., Wang, S. et al. (2004). *A road map on human-human interaction and fine-function collaboration in collaborative integrated design environments*. Xiamen, China. Springer Verlag, pp. 59-65. ISBN 0-7803-7941-1.

Ignat, C., Norrie, M. (2002). *Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems*. New Orleans, Louisiana. IEEE. IEEE Distributed Systems Online. ISBN 1541-4922.

Ignat, C., Norrie, M. (2003). *Customizable Collaborative Editor Relying on treeOPT Algorithm*. Helsinki, Finland. Kluwer Academic Publishers, pp. 315-334. ISBN 1-4020-1573-9.

Imine, A., Molli, P., Oster, G. et al. (2003). *Proving Correctness of Transformation Functions in Real-Time Groupware*. Helsinki, Finland. Kluwer Academic Publishers, pp. 277-294. ISBN 1-4020-1573-9.

Ionescu, M., Marsic, I. (2000). *An arbitration scheme for concurrency control in distributed groupware*. Retrieved July 15, 2007 from: http://www.caip.rutgers.edu/disciple/Publications/cscw2000wces.pdf.

Kung, H. T., Robinson, J. T. (1981). *On Optimistic Methods for Concurrency Control*. New York, NY, USA. ACM Press, pp. 213-226. Volume 6, Issue 2. ISBN 0362-5915.

Lamport, L. (1978). *Time, clocks and the ordering of events in a distributed system*. New York, NY, USA. ACM Press, pp. 558-565. Volume 21, Issue 7. ISBN 0001-0782.

Lauwers, J. C. (1990). *Collaboration transparency in desktop teleconferencing environments*. Stanford, California, USA. PhD Thesis. Stanford University.

Li, D., Li, R., Yu, Y. et al. (2003). *Using Familiar Single-User Editors for Collaborative Editing*. Hawaii. IEEE Computer Society, p. 10. ISBN 0-7695-1874-5.

Lu, J., Li, R., Li, D. (2004). *A state difference based approach to sharing semi-heterogeneous single-user editors*. Retrieved July 15, 2007 from: http://dsonline.computer.org/portal/cms_docs_dsonline/dsonline/topics/collaborative/events/iwces-6/Lu.pdf.

Molli, P. Skaf-Molli, H., Oster, G., Jourdain, S. (2002). *Sams: Synchronous, asynchronous, multisynchronous environments*. Rio de Janeiro, Brazil. IEEE, pp. 80-84. ISBN 85-285-0050-0.

Myers, E. W. (1986). *An O(ND) difference algorithm and its variations*. Algorithmica. Volume 1, Issue 1. Springer Verlag, pp. 251-266.

Nauer, P. et al. (1963). *Revised Report on the Algorithmic Language ALGOL 60*. The Computer Journal. Volume 5, Number 4, pp. 349-367.

Peinl, P. (1987). *Synchronisation in zentralisierten Datenbanksystemen*. Informatik-Fachberichte.

Springer-Verlag, p. 227.

Raynal, M., Singhai, M., et al. (1996). *Logical Time: Capturing Causality in DistributedSystems*. IEEE Computer. Volume 29, Issue: 2. IEEE, pp. 49-56.

Ressel, M., Nitsche-Ruhland, D., et al. (1996). *An integrating, transformation-oriented approach to concurrency control and undo in group editors*. Boston, Massachusetts, USA. ACM Press, pp. 288-297. ISBN 0-89791-765-0.

Rosenkrantz, D. J., Stearns, R., Lewis, P. (1978). *System Level Concurrency Control for Distributed Database Systems*. New York, NY, USA. ACM Press, pp. 178-198. Volume 3, Issue 2. ISBN 0362-5915.

Suleiman, M., Cart, M., et al. (1997). *Serialization of Concurrent Operations in a Distributed Collaborative Environment*. Phoenix, Arizona, USA. ACM Press, pp. 435-445. ISBN 0-89791-897-5.

Sun, C., Chen, D. (2002). *Consistency maintenance in real-time collaborative graphics editing systems*. New York, NY, USA. ACM Press, pp. 1-41. Volume 9. ISBN 1073-0516.

Sun, C., Ellis, C. (1998). *Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements*. Seattle, Washington, USA. ACM Press, pp. 59-68. ISBN 1-58113-009-0.

Sun, C., Jia, X., et al. (1998). *Achieving Convergence, Causality-preservation and Intention-preservation in Real-time Cooperative Editing Systems*. ACM Press, pp. 63-108. Volume 5, Issue 1. ISBN 1073-0516.

Sun, C., Yang, Y., Zhang, Y., Chen, D. (1996). *Distributed concurrency control in real-time cooperative editing systems*. Singapore. Springer Verlag, pp. 84-95. ISBN 3-540-62031-1.

Vidot, N., Cart, M., Ferrie, J., Suleiman, M. (2000). *Copies convergence in a distributed real-time collaborative environment*. Philadelphia, Pennsylvania, USA. ACM Press, pp. 171-180. ISBN 1-58113-222-0.

Xia, S., Sun, D., Sun, C. et al. (2004). *Leveraging single-user applications for multi-user collaboration: the CoWord approach*. Chicago, Illinois, USA. ACM Press, pp. 162-171. ISBN 1-58113-810-5

Bibliography

Arciniegas, F. (2002). *XML Developer's Guide*. Poing: Franzis. 600 pages. ISBN 3-77237-703-3.

Bernstein, P. A., Newcomer, E. (1997). *Principles of Transaction Processing*. San Francisco: Morgan Kaufmann. 358 pages. ISBN 1-558604-154.

Bernstein, P., Goodman, N., Hadzilacos, V. (1987). *Concurrency Control and Recovery in Database Systems*. Munich: Addison-Wesley. 370 pages. ISBN 2-01107-155.

Date, C. J. (2003). *An Introduction To Database Systems*. Munich: Addison-Wesley. 1024 pages. ISBN 3-21197-844.

Haerder, T. (2001). *Datenbanksysteme: Konzepte und Techniken der Implementierung*. 2. Edition. Berlin: Springer Verlag. 580 pages. ISBN 3-540-65040-7.

Kernigham, B. W., Ritchie, D. M. (1988). *The C Programming Language*. Prentice Hall. 274 pages. ISBN 0-13-110362-8.

Landau, D. P., Binder, K. (2000). *A Guide to Monte Carlo Simulation in Statistical Physics*. Cambridge: Cambridge University Press. 389 pages. ISBN 5-21653-665.

Michel, Th. (1999). *XML Kompakt. Eine Einführung*. Munich: Carl Hanser. 240 pages. ISBN 978-3446218246.

Van der Vlist, E. (2003). *XML Schema*. Cologne: O'Reilly. 448 pages. ISBN 3-89721-345-1.

Walmsley, P. (2001). *Definitive XML Schema*. London: Prentice Hall PTR. 560 pages. ISBN 1-30655-678.

Wyke, R., Watt, A. (2002). *XML Schema Essentials*. New York: Wiley. 304 pages. ISBN 4-71412-597.

Glossary

AC – The Awareness Controller component.

ACCI – The Abstract Concurrency Controller Implementation.

ADSL – Asynchronous Digital Subscriber Line. A data communication technology.

AOP – Aspect Oriented Programming.

AOSD – Aspect Oriented Software Design.

API – Application Programmer's Interface.

AW – The Awareness Widget component.

BNF – Backus Naur Form.

CAD – Computer Aided Design.

CC – The Concurrency Controller component.

CD – Character Data.

CEFX – Collaborative Editing Framework for XML.

CMAX – Consistency Maintenance Algorithm for XML.

COV – Concurrent Operations Vector.

CRC – Class Responsibility Collaboration Cards.

CRH – Conflict Resolution Hint.

CRHM – Conflict Resolution Hint Map.

CRM – Customer Relationship Management.

CRP – Conflict Resolution Provider.

CRS – Conflict Resolution Scheme.

CSCW – Computer Supported Cooperative Work.

CSV – Character Separated Values.

CTS – Conflict Type Specification.

CVS – Concurrent Versions System.

DA – The DOM Adapter component.

DCRP – The Default Conflict Resolution Provider component.

DMCS – Distributed Memory Consistency System.

DNS – Domain Name System.

DOM – Document Object Model.

DSL – Digital Subscriber Line. See ADSL.

DTD – Document Type Definition.

EBNF – Enhanced Bacchus Naur Form.

FIFO – First In First Out.

GUI – Graphical User Interface.

HB – History Buffer.

HTML – Hyper-Text Markup Language.

HTTP – Hyper-Text Transfer Protocol.

ICT – Intelligent Collaboration Transparency.

IDE – Integrated Development Environment.

IP – Internet Protocol.

IPTV – Internet Protocol Television.

ISO – International Standards Organisation.

JAMM – Java Applets Made Multi-user.

JAR – Java Archive.

JAXB – Java XML Binding.

JOS – Java Object Serialisation.

JRE – Java Runtime Environment.

JRMP – Java Remote Method Protocol.

JXTA – Juxtapose. Java Based Peer-To-Peer framework.

LAN – Local Area Network.

MIME – Multi-purpose Internet Mail Extensions.

NAT – Network Address Translation.

NC – Network Controller component of CEFX.

OASIS – Organisation for the Advancements of Structured Information Standards.

P2P – Peer-To-Peer.

PDA – Personal Digital Assistant.

PDM – Product Data Management.

PLM – Product Live cycle Management.

RCO – Resolvable Conflict Operations.

RFC – Request For Comment.

RMI – Remote Method Invocation.

RTCE – Real-Time Collaborative Editing.

RTS – Read Timestamps.

SCP – Session Control Protocol.

SGML – Standardised General Mark-up Language.

SMUX – Multiplexing Protocol.

SV – State Vector.

SVG – Scalable Vector Graphics.

SVS – State Vector Swapping.

TA – Transparent Adaptation.

TCP – Transmission Control Protocol.

UDP – User Datagram Protocol.

URI – Universal Resource Locator.

UUID – Universally Unique Identifier.

VDSL – Very High Speed Digital Subscriber Line. See ADSL.

VNC – Virtual Network Computing.

VoIP – Voice over Internet Protocol.

VPN – Virtual Private Network.

VRML – Virtual Reality Modelling Language.

W3C – World Wide Web Consortium.

WDL – Wait Depth Limited.

WebDAV – Web-based Distributed Authoring and Versioning.

Wiki – WikiWiki (Hawainese for quick,quick). A collaborative website system.

WTS – Write Timestamps.

WYSIWID – What You See Is What I Do.

WYSIWIS – What You See Is What I See.

X3D – Extensible 3D.

XHTML – Extensible Hyper-Text Markup Language.

XML – Extensible Markup Language.

Appendix

Detailed class diagrams

The following class diagrams show in detail the classes of the dynamic simulation software from chapter 2.4. The classes shown here correspond to the classes shown in figure 2.15.

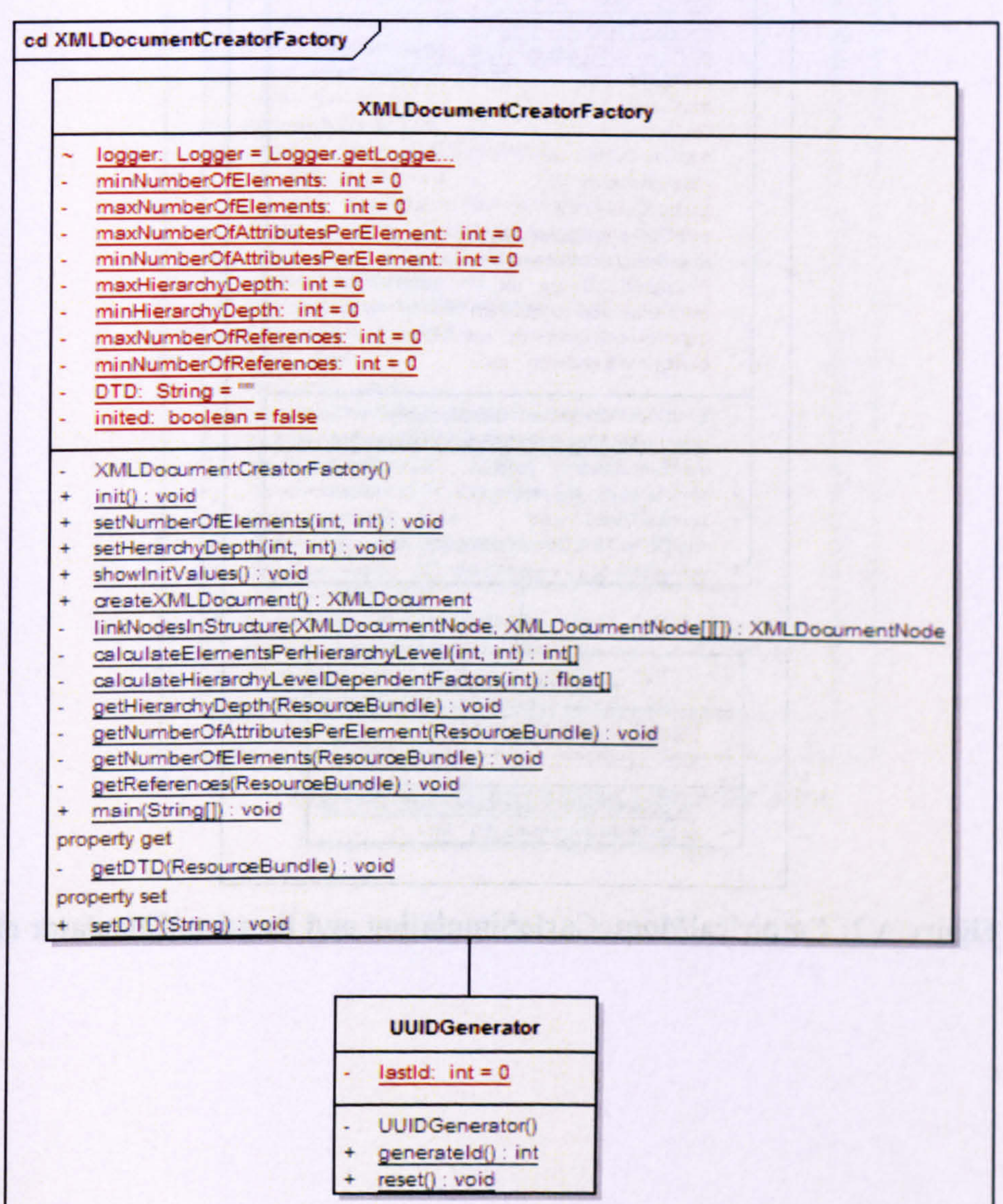

Figure A.1: XMLDocumentCreatorFactory and UUIDGenerator classes

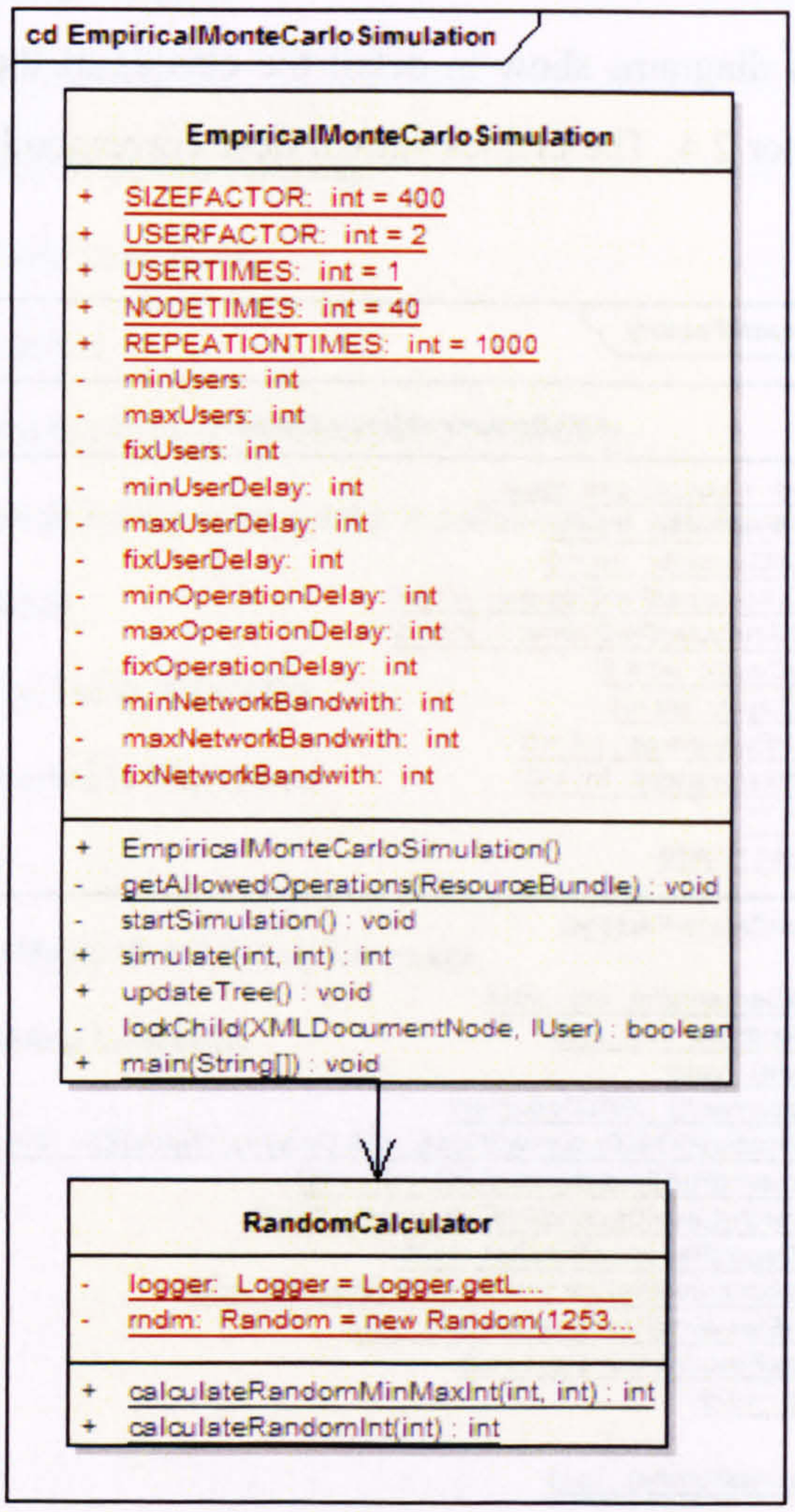

Figure A.2: EmpiricalMonteCarloSimulation and RandomCalculator classes

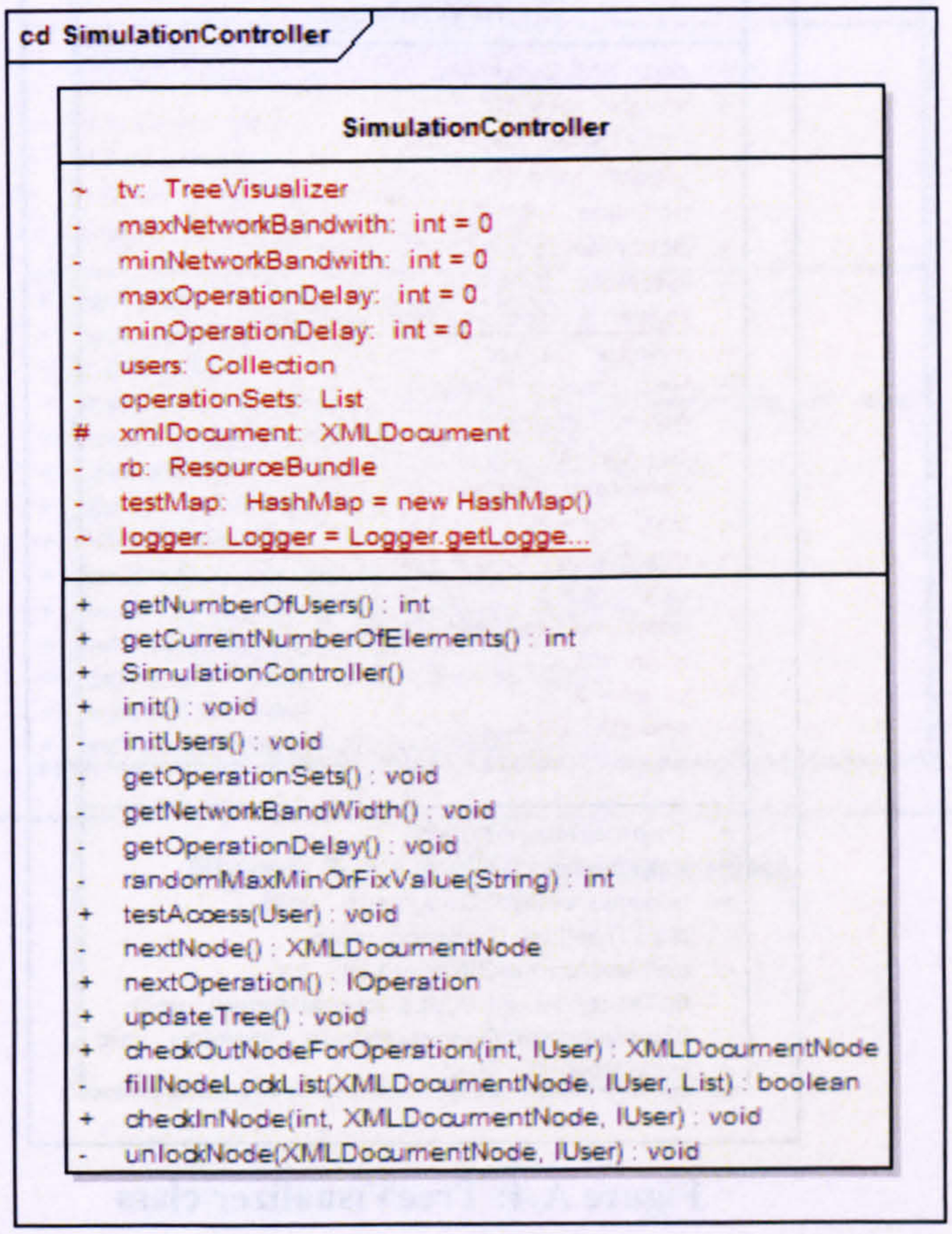

Figure A.3: SimulationController class

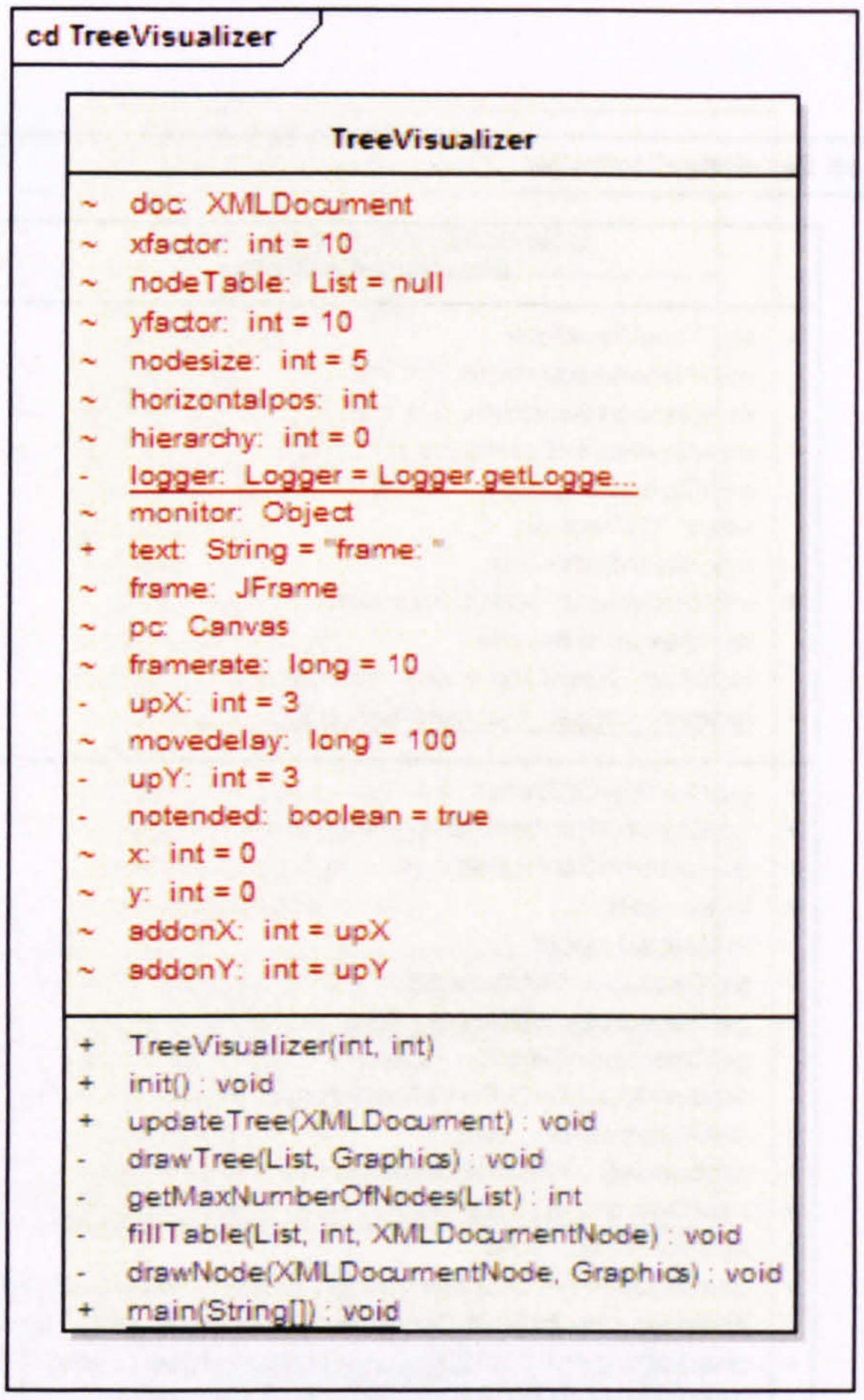

Figure A.4: TreeVisualizer class

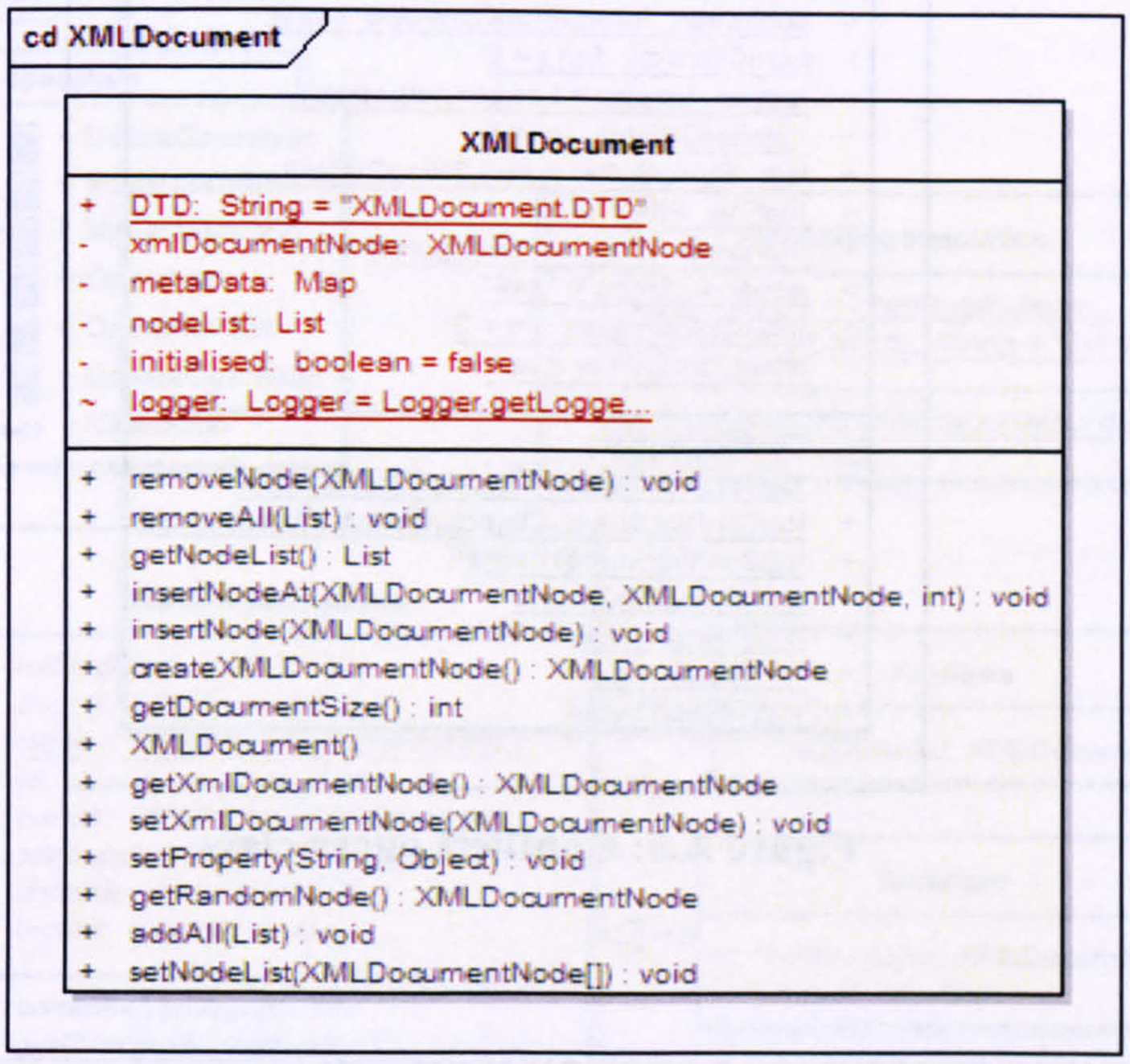

Figure A.5: XMLDocument class

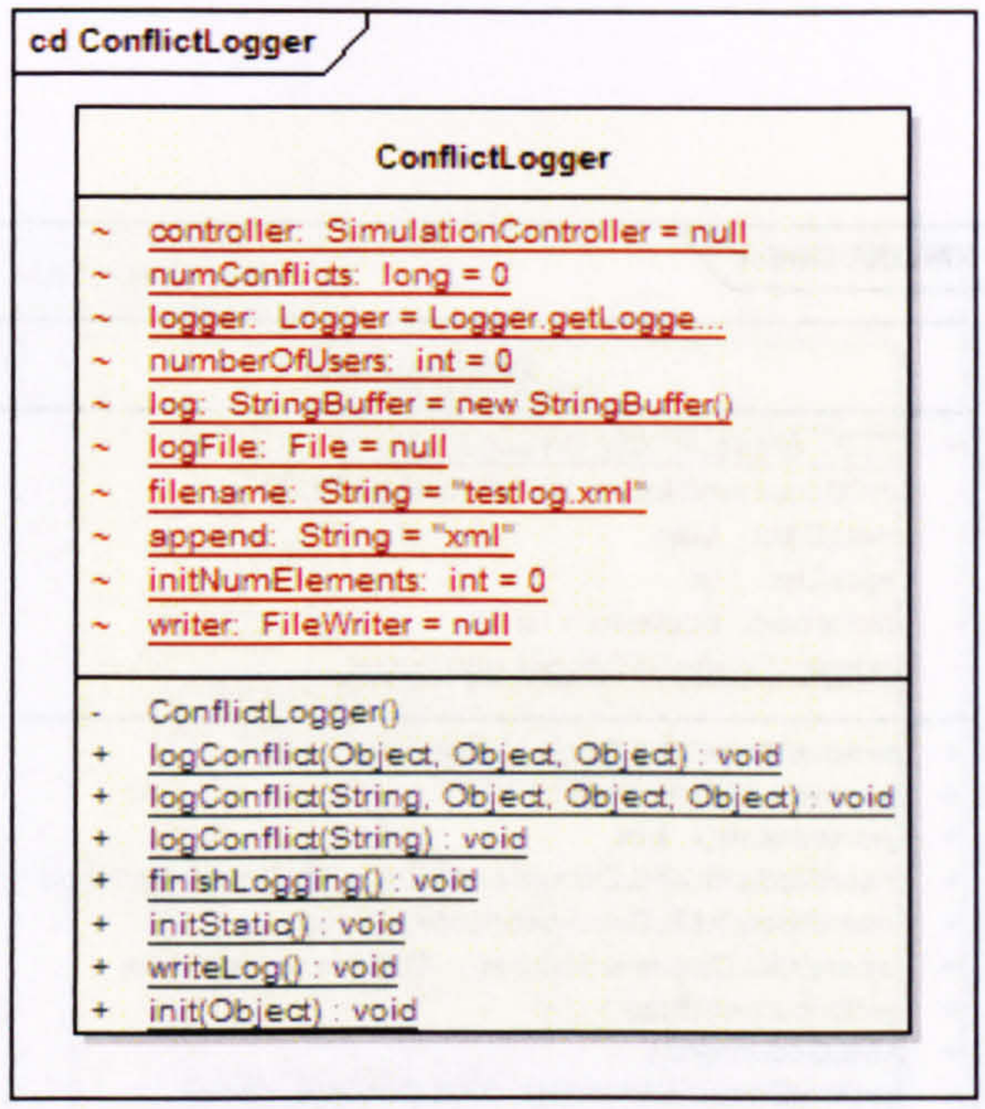

Figure A.6: ConflictLogger class

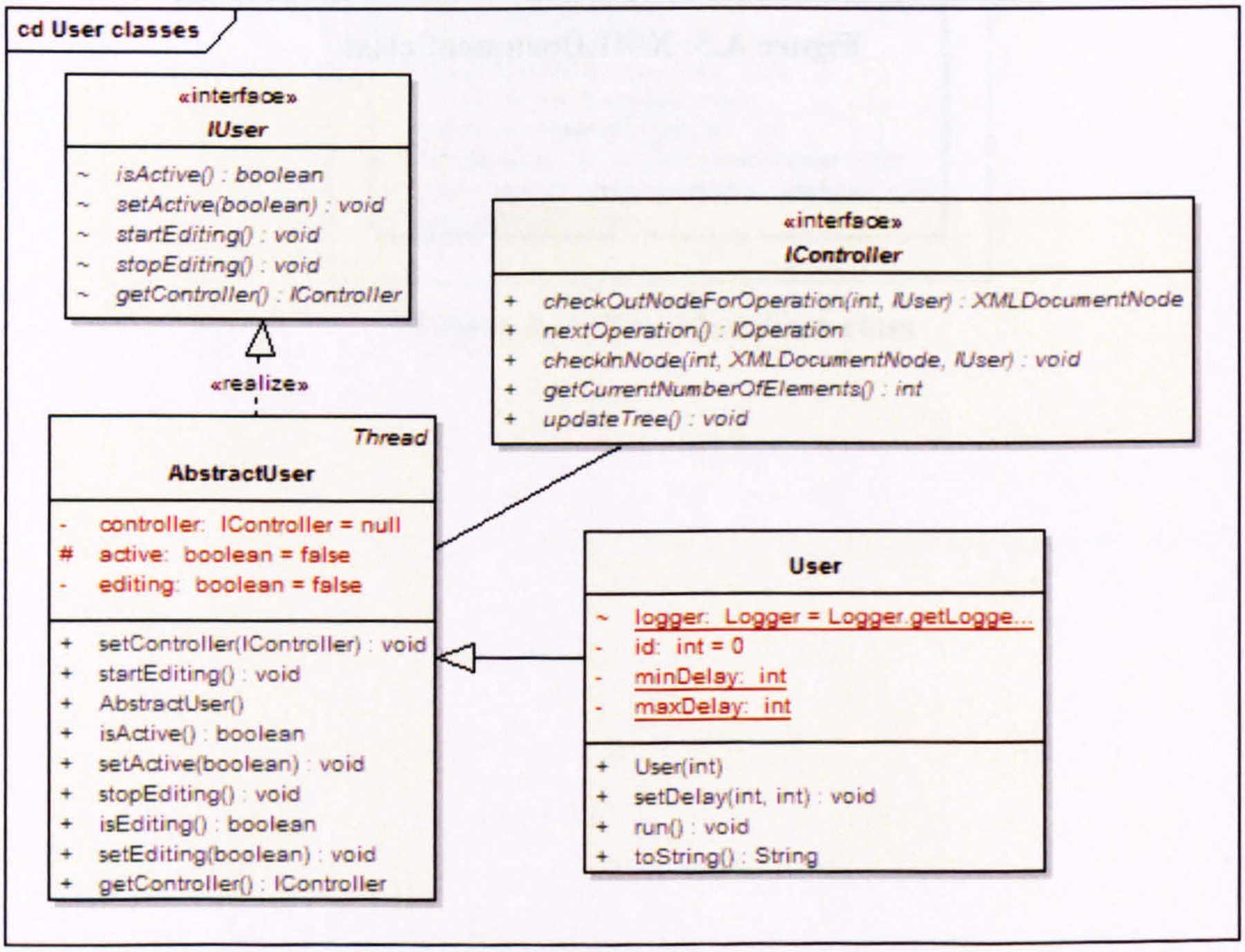

Figure A.7: User classes and interface and their relation to the IController interface

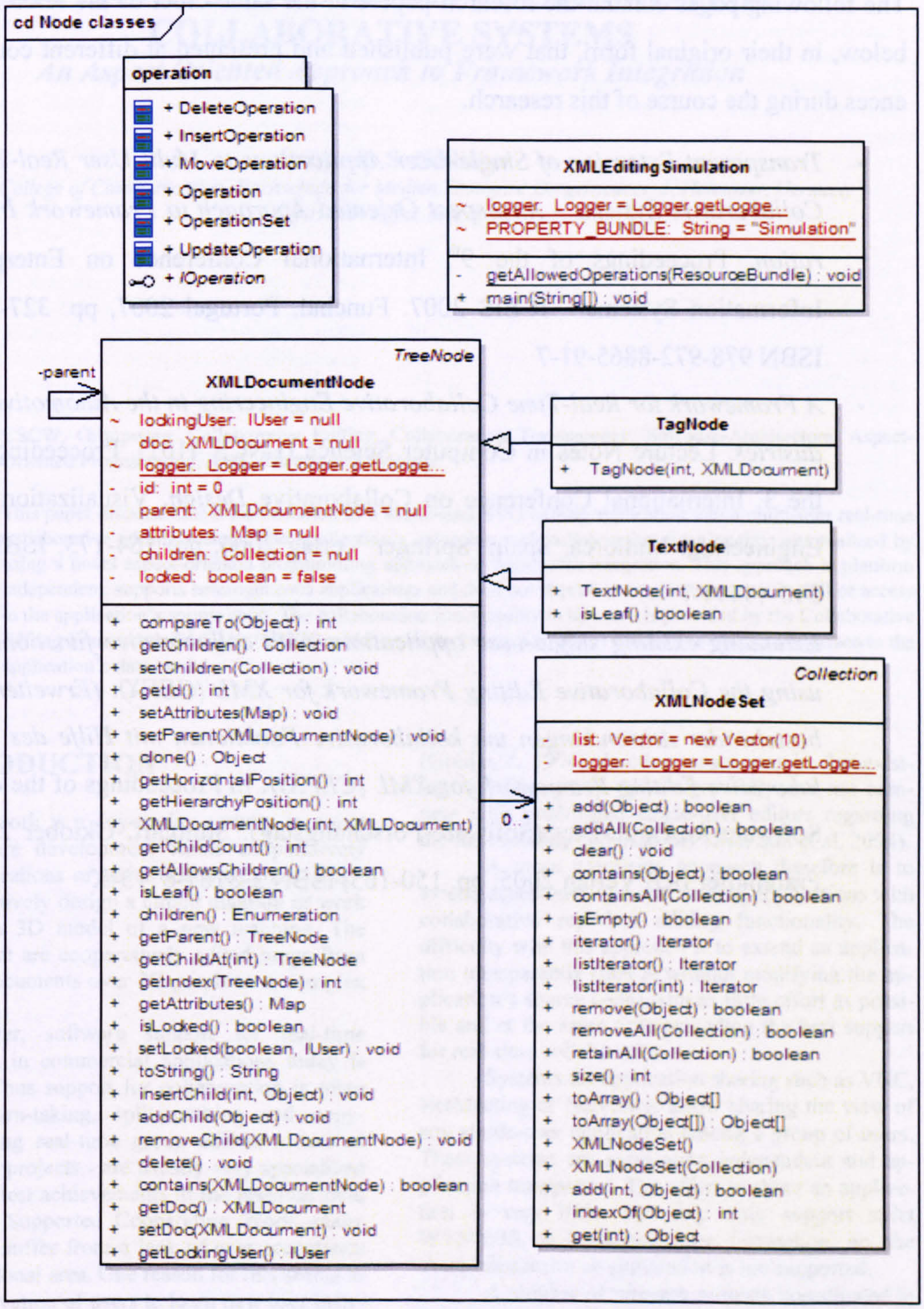

Figure A.8: Node classes and their relation, the operation package and the XMLEditingSimulation class

Published research papers

The following pages contain the research papers in the same order as the references below, in their original form, that were published and presented at different conferences during the course of this research.

- *Transparent Extension of Single-User Applications to Multi-User Real-Time Collaborative Systems - An Aspect Oriented Approach to Framework Integration.* Proceedings of the 9th International Conference on Enterprise Information Systems – ICEIS 2007. Funchal, Portugal 2007, pp. 327-334. ISBN 978-972-8865-91-7
- *A Framework for Real-Time Collaborative Engineering in the Automotive Industries*. Lecture Notes in Computer Science (LNCS 4101). Proceedings of the 3. International Conference on Collaborative *Design*, Visualization and Engineering. Mallorca, Spain. Springer Verlag 2006, pp. 164-173, ISBN 3-540-44494-7
- *Extending existing single-user applications with collaborative functionality using the Collaborative Editing Framework for XML (CEFX). (Erweiterung bestehender Anwendungen um kollaborative Funktionen mit Hilfe des Collaborative Editing Framework for XML (CEFX))*. In Proceedings of the doIT Software-Research-Day (Software-Forschungstag). Stuttgart. Oktober 2004. Fraunhofer IRB Verlag 2005, pp. 150-165, ISBN 3-8167-6715-X

TRANSPARENT EXTENSION OF SINGLE-USER APPLICATIONS TO MULTI-USER REAL-TIME COLLABORATIVE SYSTEMS

An Aspect Oriented Approach to Framework Integration

Ansgar R. S. Gerlicher
London College of Communication, Hochschule der Medien, Stuttgart, Dingelstedtstr. 5, Hannover, Germany
a.gerlicher1@lcc.arts.ac.uk

Keywords: CSCW, Groupware, Collaborative Editing, Collaboration Transparency, Software-Architecture, Aspect-Oriented Programming, XML, Distributed-Systems.

Abstract: This paper discusses the transformation of a single-user SVG editing application into a multi-user real-time collaborative editing system. The application's extension with collaboration functionality was realized by using a novel aspect-oriented programming approach to framework integration. This approach is platform independent, supports heterogeneous applications and does not require an application specific API or access to the application's source code. The collaboration functionality in this case is provided by the Collaborative Editing Framework for XML (CEFX) which uses the Document Object Model as a standard interface to the application's data model.

1 INTRODUCTION

Cooperative work is a day-to-day activity in many areas. Software development teams cooperatively develop applications or write documentation. Engineers cooperatively design a circuit diagram or work together on a 3D model of a new machine. The documents that are cooperatively edited range from simple text documents over 2D graphics to complex 3D models.

However, software support for real-time group editing in commercial applications today is uncommon. Thus support for collaboration is often limited to turn-taking, split-combine and copy-merge. Existing real-time group editors - derived from research projects - are usually very specialized and despite latest achievements in the research field of Computer Supported Cooperative Work, many such systems suffer from a lack of user acceptance in the professional area. One reason for this seems to be a low motivation of users to learn new user interfaces and application functions if they can not see their personal benefit in the collaboration features (Grudin, J., 1994). Another reason may be that existing real-time group editors generally can not compete with established single-user editors regarding the functionality and usability (Xia, Sun et al. 2004).

A more promising approach therefore is to extend accepted single-user editing applications with collaborative real-time editing functionality. The difficulty with this approach is to extend an application transparently (that is without modifying the application's source code) with as little effort as possible and at the same time providing the best support for real-time collaboration.

Systems for application sharing such as VNC, NetMeeting or Netviewer allow sharing the view of any single-user application among a group of users. These systems are application independent and application transparent. The effort to share an application is very little, but they only support strict WYSIWIS. Independent user interaction on the shared document or application is not supported.

A number of research projects investigated in how to transparently extend single-user applications with collaborative editing functionality (Xia, Sun et al., 2004, Li, Li, Yu and Yang, 2003, Begole, 1999).

Their approaches support real-time collaboration and relaxed WYSIWIS. However, these approaches either depend on an application's programming interface (API) or require translating operating system events into meaningful editing operations. This is labour intensive and the integration code has to be rewritten for each new application that is to be extended.

In this paper, we propose a novel approach to extending a single-user application with collaborative real-time editing functionality. Our approach supports relaxed WYSIWIS and heterogeneous applications. An application is thereby extended without modification of the source code and without being dependent on an application API or on operating system event translation. Our approach only assumes a standard interface to the applications data model and a runtime environment that supports introspection. We use the functionality of the Collaborative Editing Framework for XML (CEFX) in order to extend an application (Gerlicher, 2006). CEFX, among other things, takes care of synchronising the shared document. This paper discusses how we applied the concepts of aspect-oriented programming (AOP) in order to integrate CEFX into an existing single-user editing application.

The paper is structured as follows. In the next section, the different approaches of recent research projects are discussed and compared with our approach. Section 3 discusses our approach, the CEFX software architecture, the implementation and how awareness support can be achieved. Section 4 discusses the requirements for this approach and then conclusions are drawn in the last section.

2 RELATED WORK AND GOALS

Xia, Sun et al. propose the Transparent Adaptation (TA) approach for the extension of single-user applications with collaboration functionality (Xia, Sun et al., 2004). In CoWord, they have extended the Microsoft Word application transparently by making use of the Microsoft application and execution environment APIs.

The TA approach requires each application to be adapted before being shared. The adaptation of an application requires the developer to have a detailed knowledge of the application and execution environment specific API. Additionally an interpretation of the user actions in relation to the current application contexts is required. Operational Transformation (OT) (Chen, Sun et al., 1998) is used for the concurrency control of a shared Word document. This requires to map each operation executed on the application's data model into an operation that can be processed by the OT concurrency control mechanism. This is the responsibility of the Collaboration Adapter in CoWord. However, mapping user actions to OT operations can become complex and requires that the application's data model supports positional addressing of objects, which may not be feasible for complex 3D modelling applications. These limitations of CoWord are not inherent to the OT approach, but to come from the design choices made concerning the integration of concurrency control into an application.

A similar approach is proposed by Li et al. called Intelligent Collaboration Transparency (ICT) (Li,Li;Yu and Yang, 2003). The focus of their work is on sharing heterogeneous applications of the same application family. They propose a system that allows extending single-user applications such as GVim and MS Word. For each application a so called ICT agent is implemented that captures events from the operating system and the application, translates them into semantic operations and then transmits those to the other collaborating sites, where the events are replayed in the form of a sequence of editing events. Their event capture and replay mechanism makes intensive use of application and operating system specific APIs and thus suffers from the same problems as the TA approach in terms of implementation complexity. Additionally the complexity is increased by supporting heterogeneous applications. This requires a formalisation of application semantics in order to be able to translate the user actions of one application to the relating user actions of another application.

The high implementation effort was one of the reasons for the second generation of the ICT project, ICT2. In contrast to their previous work, ICT2 does not attempt to intercept and understand the operating system level events. Instead it uses an adapted version of the diffing algorithm (Myers, 1986) to derive the editing sequences between document states (Lu and Li, 2004). However, this new approach is not suited for fine-grained real-time group editing such as TA and ICT, because of its limitations in terms of performance. The support for heterogeneous applications is limited to those applications that have the same writing style. Sharing a document between for example Latex and Word is not supported. The diff algorithm that is applied supports determining the differences of text documents only. In order to support structured and formatted documents, more sophisticated diffing algorithms would be required (Li and Lu, 2006).

Begole (Begole, 1999) proposes a different approach. The Flexible JAMM (Java Applets Made Multi-User) project extends a single-user application

by replacing selected components of it with multi-user versions. The basic idea of this approach is not to use an application or operating system specific API for the integration of collaboration function, but the Java Swing API. This has the advantage that the components which are replaced are well known and it is not required to implement a translation layer as in the TA approach or convert user actions into application semantic commands or API calls as in ICT. The disadvantage is that the set of applications that are suitable for an extension is restricted to Java Swing based applications.

To summarise, all approaches use a certain API in order to extend a single-user application. TA and ICT both use an API on operating system and on application level. JAMM uses an API on the level of the runtime's graphical user interface (GUI) library. The first two approaches face the problem of implementation complexity for each new application that is to be extended. The JAMM approach has the problem of being dependent on the fulfilment of certain runtime requirements.

The goal of our approach is to reduce the complexity of integrating a collaboration framework into a single-user application and provide a solution that is more general, supporting many different types of applications. We argue that this can be achieved by using aspect-oriented programming and concentrating on the application's data model instead of an application, operating system or GUI library API.

The application data model describes how data is represented and used. For example in a text editing application, the data model represents the text that is edited. The structure of the data model can thereby be different to its visual representation. One aspect of an application is the manipulation of the data model. The code for updating or querying the data model can be distributed within the entire application. In the terminology of AOP such aspects are called crosscutting concerns. In our approach, we identify these crosscutting concerns or system-level-concerns within an application and define advices that then create events for the underlying collaboration framework in order to synchronise the data model between the different sites. This has the advantage that once developed advices can be reused for all applications that use the same methods for the manipulation of their data model. This is for example the case for all applications using the Document Object Model (DOM) as a standard interface for the manipulation of XML content. Although new advices have to be developed for applications that use other methods for the data model manipulation, we assume that the implementation effort is low compared to other methods.

3 THE AOP APPROACH

For extending a single-user application with collaborative real-time editing functionality, the Collaborative Editing Framework for XML (CEFX) was used. CEFX is a collaborative real-time system specialised on XML based applications. It satisfies the collaborative requirements of communication, group awareness, session management and concurrency control (Pichiliani and Hirata, 2006).

We selected the GLIPS Graffiti editor (GLIPS Graffiti Editor) as single-user application for the transparent integration of CEFX. The GLIPS Graffiti editor is a cross-platform SVG graphics editor developed by ITRIS (ITRIS). It enables to create regular SVG files. As GLIPS is a Java application, it was necessary to use an aspect-oriented extension to the Java programming language. We used AspectJ for the development of the required aspects.

Aspect-oriented programming (AOP) is a programming paradigm for the separation and encapsulation of concerns, especially crosscutting concerns, within a software. The most popular AOP language is AspectJ (The AspectJ Project), developed by Gregor Kiczales et al. at Xerox PARC (PARC).

In order to encapsulate crosscutting concerns at one place, so called aspects are defined which are then integrated into the software not earlier then at compile time. Using AOP implementations that support byte-code weaving (AspectJ), allows to extend applications without access to their source code. An aspect can alter the behaviour of the base code (the non-aspect part of a program) by applying advices (additional behaviour) at various join points (points in a program) specified in a quantification or query called a pointcut (that detects whether a given join point matches). An aspect can also make binary-compatible structural changes to other classes, like adding members or parents. (Aspect-oriented programming).

The following section describes the relevant parts of the CEFX software architecture. Section 3.2. discusses the implementation of the advices and in section 3.3 the integration of awareness mechanisms is discussed.

3.1 Software Architecture

CEFX is based on a hybrid software architecture, which is a mixture of a centralised and a replicated architecture. The central server side is responsible for the session handling and management of the shared documents. When a client connects to the server in order to work on a certain document, the server checks if a session for that requested document is already open and connects the client to it.

The client then retrieves a copy of the shared document from the server. Each operation that is executed at a client site is executed locally first and then propagated to each other client in a session and to the server site. This guarantees good response times.

CEFX is composed of a set of plug-in components that are responsible for the different aspects of a collaboration system. The flexible plug-in architecture of CEFX supports the extension of the framework itself by providing new plug-in components. The figure below depicts a simplified model of the internal structure of the client part of CEFX.

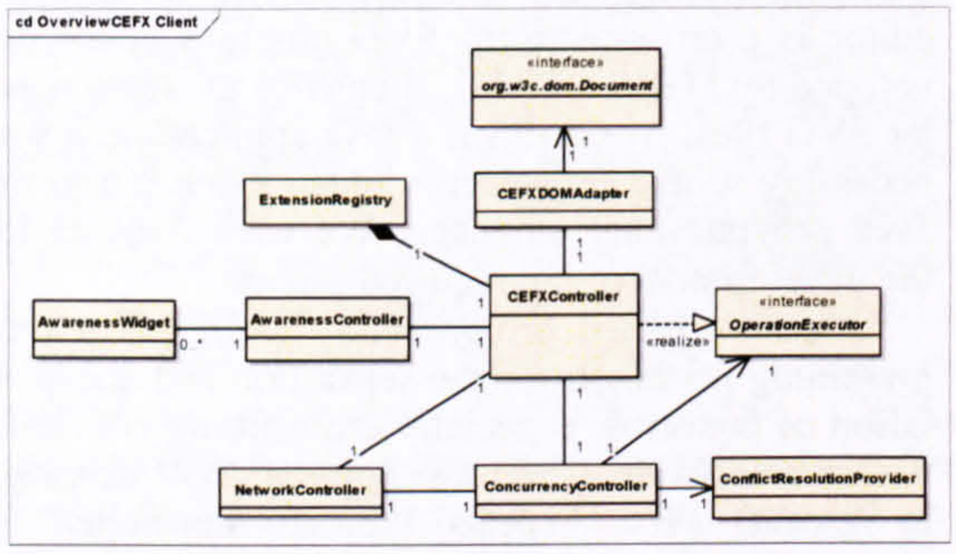

Figure 1: CEFX client components

The CEFX Controller is the central controller component of the framework. It owns a Extension Registry (ER) which contains information on plug-in components that need to be instantiated. The CEFX Controller processes all framework events and delegates them to the corresponding components.

The Network Controller (NC) is responsible for sending and retrieving information over the network The Concurrency Controller (CC) at each client and the server site takes care of the synchronisation of the shared documents and handles conflicts. The concurrency control algorithm used was specifically developed for the synchronisation of XML documents. It follows the same principles of causality and convergence as the algorithms described in Davis, Sun et al., 2002, Ignat and Norrie, 2002 and Molli et al. 2002. In contrast to the discussed systems, universal unique ids (UUIDs) are used for addressing nodes and operational transformation is not applied for preserving user intentions. However, a detailed explanation of the used consistency maintenance algorithm in CEFX goes beyond the scope of this paper.

In the case of a conflict the Conflict Resolution Provider (CRP) provides hints to the concurrency controller on how the conflicting operations should be treated.

Furthermore each client site has an Awareness Controller (AC) that is responsible for the propagation of awareness events, such as mouse selection events. It delegates awareness information to the corresponding awareness widgets which are responsible for the visualisation of these events.

The CEFX DOM Adapter (DA) is our entry point to the framework. At the beginning of each collaborative session, when a document is opened by the user, the DA is provided with a reference to the Document Object Model (DOM) of the application. This is done by one of the advices that are called if a certain join point within the application is executed. For each local modification of the DOM, the DA creates an operation which is then executed and propagated to the other sites. Incoming operations are - after passing different synchronisation steps - eventually executed directly on the DOM of the application, as if they were executed locally by a user action.

3.2 Implementation

When extending the GLIPS Graffiti editor, the crosscutting concerns that we were interested in is the modification of the applications data model. The first step was to identify the relevant join points that are executed, when the data model is modified. User operations such as drawing a line, changing the colour of an object or deleting a object modify the applications data model.

The GLIPS Graffiti editor is used to create and edit Scalable Vector Graphics (SVG). As SVG is an XML document format, GLIPS uses an XML document internally as data model. For the modification of the XML document, GLIPS makes use of the DOM API. For rendering the SVG graphics to the screen, the Apache Batik library is used (Batik SVG Toolkit). Batik provides its own implementation of the DOM which complies to the W3C Document Object Model specification (W3C Document Object Model).

This simplified the identification of the relevant join points. All calls to functions defined by the W3C DOM API were possible candidates for a relevant join point.

The next step was to write an Aspect class that encapsulates the crosscutting concerns at one place. The Aspect class is similar to a Java class and can contain normal Java code and additionally AspectJ components. A simple example:

```
public aspect DOMAccessAspect {
...
}
```

The defined Aspect is then weaved into the applications bytecode at compile time. The applications source code is not modified. The Aspect class defines point cuts that are executed by AspectJ when the matching join points are reached within the application.

We identified calls to the following W3C DOM API methods as most relevant to the modification of the GLIPS data model:

```
Node appendChild(…)
Node insertBefore(…)
Node removeChild(…)
void setAttributeNS(…)
void setAttribute(…)
```

The methods `appendChild()` and `insertBefore()` are called, whenever a node is added to the document. This is the case for example if the user draws a line in the SVG document. The method `removeChild()` is called, whenever a node is removed from the document. That is the case if the user deletes an object from the document. The methods `setAttributeNS()` and `setAttribute()` are called if for example the user changes the colour of an object. In SVG the colour information of an object is contained in the value of the `style` attribute.

The figure below shows a scenario of what happens, if for example the colour of an object is changed. Whenever the method `setAttribute()` on an element node of the XML document is called, the call is intercepted by AspectJ. Instead of directly executing the code of the DOM implementation provided by Batik, the defined pointcut of our aspect class selects the relevant join point and the specific advice is applied. The advice then delegates the call to the DA. The DA creates an update operation which is handled by the CEFX Controller and asynchronously propagated to all sites of the current editing session by the NC. The operation is instantly executed on the local element node and the attribute value is updated.

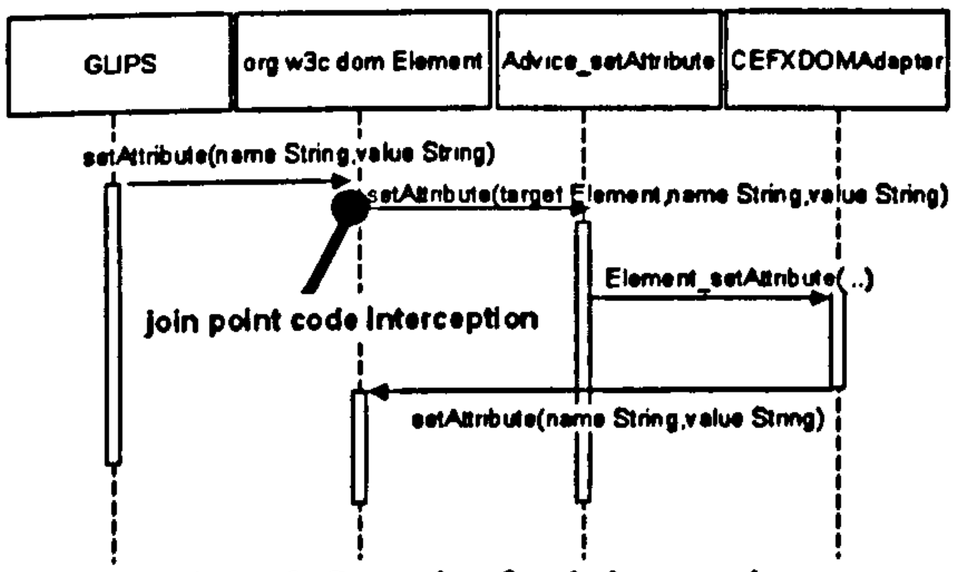

Figure 2: Scenario of code interception

Pointcuts pick out interesting join points in the execution of a program. These join points can be for example method invocations and executions. An AspectJ pointcut definition gives a name to a pointcut. The code snippet below shows how the pointcut of the above scenario is defined in AspectJ.

```
…
pointcut setAttributePC(Element p,
String attr, String value):
target(p) && args(attr,value) &&
call (
void setAttribute(String,String))
&& !within(DOMAccessAspect);
…
```

This pointcut picks out all join points matching a call to a method with the same signature and parameters as given in the `call()` statement. For each pointcut an advice is defined, that contains the code that is to be executed if the pointcut is met.
For each of the relevant pointcuts, we defined an advice. The advice, that is executed for the above pointcut is shown in the following code snippet.

```
…
void around(Element p, String attr,
String value):
setAttributePC(p,attr,value)
{
Advices.setAttribute(p,attr,value);
}
…
```

The static method `setAttribute()` of the class `Advices` is called here. The class `Advices` is a helper class containing the advices code. This was done for clarity reasons and in order to separate the Java code from AspectJ code. The code that is executed here is shown in the following code snippet.

```
…
public static void setAttribute
(Element p,String attr,
 String value)
{
 CEFXDOMAdapter doma =
 CEFXUtil.getDOMAdapter();
 if (doma != null) {
  if(doma.isCollaborationReady()) {
   doma.Element_setAttribute(p,attr,
    value);
    return;
   }
 }
 p.setAttribute(attr, value);
}
…
```

First a reference to the CEFXDOMAdapter is retrieved by calling the static method getDOMAdapter() of the CEFXUtil class, a utility class provided by CEFX. If the DA is already initialised and ready for collaboration, the call to setAttribute() is delegated to the corresponding DA method. If the DA was not properly initialised, the setAttribute() of the target object is called directly, setting the new value to the attribute. This is done for example, if the client is not connected to a collaboration session.

For all other relevant pointcuts the same procedure was applied. The following code shows an example on how the other advices were defined using anonymous pointcuts.

```
...
 Node around(Element p, Element c):
 target(p) && args(c) &&
 call(Node appendChild(Node)) &&
 !within(DOMAccessAspect)
 {
   return Advices.appendChild(p,c);
 }
...
```

Additionally to the advices for the manipulation of the data model, advices for creating or loading of a document and the initialisation of the render context are needed. Creating a new SVG document or loading and parsing of an existing document is handled by the SAXSVGDocumentFactory class provided by Batik. Rendering of an SVG document to the screen is handled by the SVGCanvas class (the render context) provided by GLIPS. When a document is opened, the DA is provided with a reference to it, in order to integrate changes from the remote sites. After the execution of a remote operation, the render context of the application is notified in order to repaint the document. For this reason, the DA is provided with a reference to the application's render context. The following advice code is executed, when a document is opened by the user.

```
...
SVGDocument
around(SAXSVGDocumentFactory fac,
String uri):
target(fac) && args(uri) &&
call(SVGDocument createSVGDocu-
ment(String)) &&
!within(DOMAccessAspect)
{
//Initalisation of the DA
 CEFXDOMAdapter da =
 new CEFXDOMAdapterImpl();
//Proving DA with factory reference
 da.setDocumentFactory(fac);
//Creating the document and
//providing DA with a reference to it
 SVGDocument doc = (SVGDocument)
 da.createDocument(uri);
...
}
...
```

The DA is also provided with a reference to the document factory. This is required for example, if a session for the opened document already exists. In this case, CEFX loads the document from the server and handles the document parsing and initialisation.

The advice for setting the render context is called when the SVGCanvas is initialised within the application. The following code illustrates how this is achieved.

```
...
after(SVGCanvas panel): target(panel)
&& call(* initializeCanvas(*))
{
 CEFXDOMAdapter da =
 CEFXUtil.getDOMAdapter();
 if (da != null) {
  da.setRenderContext(panel);
 }
}
...
```

The SVGCanvas class is derived from javax.swing.JlayeredPane and provides a method initializeCanvas(). The advice is executed after the initialisation method has been called. Thus in this case the method call is not intercepted. The advice is solely used for notifying CEFX of the initialisation and providing it with a reference to the render context.

To summarize, using aspect-oriented programming for the integration of CEFX into the GLIPS Graffiti editor did not require much programming effort. Only seven advices were needed to provide GLIPS with the basic collaboration functionality. Five of the used advices were related to the DOM API and can be reused for other applications using the DOM. One advice was specific to the Batik library and one was specific to the application. The overall performance of the application did not change noticeable.

AspectJ is one of many existing implementations of AOP. In this case for example HyperJ (HyperJ Overview) could alternatively be used. AOP implementations exist for many different languages and platforms such as Java, C#, VB.NET, JavaScript, C/C++, Lua, Python, Ruby, Perl, PHP, Common Lisp and many others.

3.3 Awareness Support

The discussed advices are used to integrate CEFX in a way that satisfies the requirements of communication, session management and concurrency control. In order to satisfy the requirement of group awareness, additional effort is necessary.

CEFX provides simple awareness widgets that allow to notify each user in a collaborative session for example on other user's mouse movements. This can help a user to get an understanding of what other users are working at. The awareness widgets are little windows, controlled by the CEFX client and independent of the extended application. The application is not aware of those widgets.

For the integration of the awareness support provided by CEFX into the GLIPS editor, additional advices can be used. Java applications make use of certain interfaces from the `java.awt.event` package in order to retrieve information on mouse clicks and mouse movements. In the following example we show how we notified CEFX of a user's mouse clicks. A new Aspect class containing advices for mouse events was developed.

```
public aspect SelectionAspect {
...
 after(MouseEvent event):
 args(event) && execution(
 void mousePressed(MouseEvent))&&
 !within(SelectionAspect)
 {
   Advices.mousePressed(event);
 }
...
```

The above code snippet shows the advice that is executed when the user presses the mouse button. The static method `mousePressed()` of the Advices helper class delegates the mouse event to the `MouseEventPropagator` component of CEFX.

```
...
public static void
mousePressed(MouseEvent event)
{
 CEFXDOMAdapter da =
 CEFXUtil.getDOMAdapter();
 if (da != null) {
 MouseEventPropagator li =
 (MouseEventPropagator)
 da.getEventPropagator(event);
 if(li!=null)
 {
  li.mousePressed(event);
 }
...
```

The `MouseEventPropagator` component is registered with the AC component of CEFX and is responsible for propagating the mouse events to the other sites, where they are displayed in the corresponding awareness widgets. The same kind of advices can be implemented for all other kinds of user events such as typing the keyboard or mouse movements. Using AOP here allows a transparent integration of awareness into the application.

4 REQUIREMENTS

AOP implementations are available for a large number of programming languages and platforms. One requirement though is that the target application is written in a language that is supported by AOP. Additionally some AOP implementations require recompilation of the application's source code in order to weave the generated aspect code into it. Other AOP implementations do not require source code. AspectJ for example supports byte-code weaving and advanced load-time weaving. This allows using AOP without access to the application's source code, which makes it suitable for the extension of commercial applications.

The integration of CEFX into the GLIPS application had the advantage that GLIPS uses the DOM for accessing it's data model. This simplified the identification of relevant join points. For applications that do not use a standard interface for modifying their data model the identification of the join points may be more difficult, but still feasible.

It is worth noticing that the support for heterogeneous applications in this approach is limited to applications using the same type of XML document. The support for real heterogeneous applications (using different XML document types) is a subject for future research.

5 CONCLUSIONS

This paper proposes a novel approach to the integration of a collaborative editing framework in order to transparently extend a single-user application with group editing functionality.

We assume that using standardised data model interfaces and aspect-oriented concepts can dramatically reduce implementation efforts in comparison to other approaches using window event translation and application specific programming interfaces. This paper shows how little the effort is to transparently extend a single-user SVG editing application

using this approach. The application was extended, by making use of the standard Document Object Model and the Collaborative Editing Framework for XML. The developed aspect-oriented programming advises are reusable and next step will be to extend a number of other single-user applications with group editing functionality.

More and more applications today use XML as a file format, for example OpenOffice, Mircosoft Word 2007 and a number of editors for other XML based file formats. If those applications make use of the DOM API internally for the modification of their data model, this will ease their extension with real-time collaboration features.

However, one aim of this research project is to provide collaboration support to existing and future applications used for the design of vehicle electrical systems in the automotive industries. Today, the SVG format is a de facto standard for the representation of circuit diagrams in this area. Other XML based document formats such as ELOG (Electrological Model) are currently under development. The development of a vehicle electrical system is a complex process requiring intensive collaboration between a number of different companies such as the OEM, the suppliers and manufacturers of the cable loom and different subcontractors, but the current applications used in this area do not provide support for real-time collaboration. Providing a system that supports real-time collaborative engineering would allow all parties to work on a single source. This could lead to a better quality and higher productivity.

REFERENCES

Chen, D. Sun, C, Jia, X. Zhang, Y. Yang, Y., 1998. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems. In *ACM Transactions on Computer-Human Interaction, Vol.5, No.1, pp.63-108*

Ignat, C. Norrie, M. C., 2002. Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems. In *ACM Proceedings. Fourth International Workshop on Collaborative Editing Systems, New Orleans, Louisiana.*

Molli, P. Skaf-Molli, H. Oster, G. Jourdain, S., 2002 Sams: Synchronous, asynchronous, multisynchronous environments. In *Proccedings of Seventh International Conference on CSCW in Design, Rio de Janeiro, Brazil*

Davis, A. Sun, C. Lu, J., 2002. Generalizing Operational Transformation to the Standard General Markup Language. In *Proceedings of ACM 2002 Conference on Computer Supported Cooperative Work, New Orleans, Louisiana, USA.*

Xia, S. Sun, D. Sun, C. Chen, D, Shen, H., 2004. Leveraging single-user applications for multi-user collaboration: the CoWord approach. In *Proceedings of ACM 2004 Conference on Computer Supported Cooperative Work, Chicago, IL USA*

Lu, J. Li, R. Li, D., 2004. A state difference based approach to sharing semi-heterogeneous single-user editors. In *Proceedings of CSCW'04 workshop on collaborative systems (IWCES-6) and application sharing systems. Chicago*

Li, D. Li, R. Yu, Y. Yang, Y., 2003. Using Familiar Single-User Editors for Collaborative Editing. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03)*

Begole, J.M.A., 1999. Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems. *Ph.D. Dissertation. Virginia Polytechnic Institute and State University, Blacksburg*

Gerlicher, A.R.S., 2006 A Framework for Real-time Collaborative Engineering in the Automotive Industries, In *Proceedings of Third International Conference on Cooperative Design, Visualization and Engineering, CDVE 2006, Mallorca, Spain*

Myers, E. W., 1986. An O(ND) difference algorithm and its variations, *Algorithmica I, pages 251-266*

Li, D. Lu, J., 2006. A Lightweight Approach to Transparent Sharing of Familiar Single-User Editors. In *Proceedings of ACM CSCW'06, Banff, Alberta, Canada*

Pichiliani, M. and Hirata, C. M., 2006. A Guide to Map Application Components to Support Multi-User Real-time Collaboration. *ITA (short paper), CollaborateCom 2006, Atlanta, Georgia, USA*

Grudin, J., 1994. Groupware and social dynamics: eight challenges for developers. *Communications of the ACM, Volume 37 ,Issue 1, pages 92 - 105*

HyperJ Overview (Tarr, P). Retrieved January 14, 2007, from http://www.alphaworks.ibm.com/tech/hyperj

GLIPS Graffiti Editor (n.d.). Retrieved January 14, 2007, from http://glipssvgeditor.sourceforge.net/

ITRIS (n.d.). Retrieved January 14, 2007, from http://www.itris.fr

PARC, Palo Alto Research Center, Inc. Retrieved January 14, 2007, from http://www.parc.xerox.com/

The AspectJ Project (n.d.). Retrieved January 14, 2007, from http://www.aspectj.org

Aspect-oriented programming - Wikipedia (n.d.). Retrieved January 14, 2007, from http://en.wikipedia.org/wiki/Aspect-oriented_programming

Batik SVG Toolkit (n.d.). Retrieved January 14, 2007, from http://xmlgraphics.apache.org/batik/

W3C Document Object Model (n.d.). Retrieved January 14, 2007, from http://www.w3.org/DOM

Electrological Model, ELOG, VDA. Retrieved January 14, 2007, from http://www.ecad-if.de/elog.html

A Framework for Real-Time Collaborative Engineering in the Automotive Industries

Ansgar R.S. Gerlicher

London College of Communication
University of Applied Science, Stuttgart
Dingelstedtstr. 5, 30655 Hannover
gerlicher@hdm-stuttgart.de

Abstract. Today, many different companies are involved in the automotive engineering process. The OEM, subcontractors and suppliers all need to collaborate and access the same data. Specialized applications are used in the process of designing vehicle electrical systems. These applications use proprietary data formats and do not support collaborative engineering. Thus collaboration methods are limited to turn-taking, split-combine and copy-merge. To become application independent and stay future-proof, a new trend is the transformation of data from the proprietary data formats to the Extensible Markup Language[1] (XML). This will allow new ways of viewing, editing and analyzing the data using new and existing applications and tools that use XML as a data model. This paper presents a novel software framework that allows easy enhancement of any such applications with the ability of collaborative real-time editing. Support for heterogeneous applications, a new flexible plug-in architecture and easy application integration are some of its key features.

Keywords: XML, Collaborative Engineering, Real-time Collaboration, Software Engineering, Groupware, CSCW, Vehicle Electrical System.

1 Introduction

The development of vehicle electrical systems is a complex and tedious process. Many different companies such as subcontractors and suppliers are involved in designing and building the cable loom of a car. In order to support the engineers, a number of applications such as Logical Cable[2], Catia[3] and LDorado[4] are used to design circuit diagrams and electrical components. The electrical components and component symbols that are used in the design are administered in a so called Cable Library System (CLS). The cable loom design is split into different modules where each module design contains one or more diagram sheets. A number of companies

---

[1] Extensible Markup Language (XML) 1.0 (Third Edition) W3C Recommendation, http://www.w3.org/XML/, 4th February 2004
[2] Logical Cable (LCable), http://www.mentor.com/products/cabling_harness/index.cfm
[3] Catia 3D, http://www.3ds.com/home
[4] LDorado, http://www.ldorado.harnesslab.de

such as the supplier or manufacturer of the cable loom, different subcontractors and the OEM (e.g. Volkswagen) are involved in the cable loom design. Thus a tight collaboration between the involved companies and the OEM is required. For the exchange of data between the OEM and the subcontractors today a so called construction data management system (CDMS) is used. Such systems are used for storing and retrieving development artifacts. For example, a copy of the CLS library (containing all electrical components that are required for designing a circuit diagram) is retrieved by the subcontractor via the CDMS. The subcontractor then uses this library to design the wiring diagrams. Problems occur here for example, when one subcontractor works with an older version of the CLS than other involved parties. When exchanging the wiring diagrams, conflicts can occur that need to be resolved manually. The process of integrating changes made by subcontractors into the overall design, the management of the design data in CDMS and an error checking procedure are time consuming and expensive. These and other problems reduce the productivity of the engineering process. To overcome these problems a synchronization mechanism would be helpful, which ensures that all companies involved in the design process, work with up to date data. In the design process a system that supports real-time collaborative engineering would allow all parties to work on a single source. Additionally the OEM would always be able to see the current status of work and check the design for errors at all times. This could lead to a better quality and higher productivity.

The development of complex applications such as a symbol editor or a wiring diagram editor is a difficult and time consuming task. Since already various single-user applications exist it would be much simpler and user friendly to enhance the existing applications with real-time collaboration functionality, instead of developing new collaborative applications. One of the advantages is, that users do not have to learn and adapt to a new application but can use their familiar application with enhanced functionality. The goal of our research is to develop a software framework that allows the enhancement of any type of single user editing system or application, using XML as a data format, with the ability of collaborative real time editing. The trend is set by the Verband der Automobilindustrie VDA (Automotive Industry Association) to use XML as the new data format for the design and exchange of vehicle electrical systems. In the automotive industry the Scalable Vector Graphics[5] (SVG) format is used to visualize and exchange wiring diagrams. Other XML based languages exist or are in development to model the logic and other data that is needed in the vehicle electrical system design. These are for example KBL[6] and Elog[7]. The KBL (Harness Description List) XML format (VDA recommendation 4964) is a subset of the AP212 ISO Specification[8] for vehicle electrical systems. It defines a data model and an XML schema for the exchange of harness design data between the OEM and the suppliers. Elog is a XML format currently under development by the VDA and will be used to represent the electro-logical model of a vehicle electrical

---

[5] Scalable Vector Graphics 1.0 Specification. W3C Recommendation, http://www.w3.org/TR/SVG/, 2001

[6] KBL, Kabelbaumliste (Harness Description List), VDA recommendation 4964, http://www.ecad-if.de/kbl.html

[7] Electrological Model, ELOG, VDA, http://www.ecad-if.de/elog.html

[8] AP212, Electrotechnical Design and Installation, ISO Standard 10303-212

system including views (sheets), electrical components, wires and connectors. As XML is becoming the standard interchange and data format not only in the automotive industries, a software system that supports real-time collaborative editing of XML documents could lead to a general solution for many application areas. The collaborative editing framework for XML (CEFX) we present in this paper, will allow enhancing applications that use XML as a data model, but is not limited to these.
The paper is structured as follows. In the next section prior work in supporting collaborative use of single-user applications and the CEFX approach are discussed. Section 3 discusses the CEFX framework architecture, the system components and how single-user applications can be extended by CEFX. In Section 4 the consistency model that is applied in the default implementation of CEFX is briefly discussed. In section 5 requirements, limitations and future work is discussed and in the last section research findings and contributions in this paper are summarized.

2 Collaborative Systems

Collaborative systems that allow the enhancement of a single-user application with collaborative functionality are typically classified into collaboration-transparent systems and collaboration-aware systems. Systems that provide methods to share a single-user application without changing the application are called collaboration-transparent. The applied methods are unknown to the application and its developers. Collaboration-aware systems integrate collaboration mechanism by changing the application so that the application is aware of these. This allows a tight integration of collaboration functionality into an existing application. The problem with the collaboration-aware approach is, that it requires access to the source code of an application which in some cases may not be possible for "of the shelf" software products. The collaboration-transparent approach does not require access to an applications source code. Two types of collaboration-transparent systems can be identified: application independent (generic) and application dependent systems. Application independent systems do not know the shared application. They work on basis of transmitting low-level input/output data such as key-strokes, mouse movements and display pixel data. Application dependent systems are used to enhance a specific application with collaboration functionality and have knowledge of the application specific data model or the application API (Application Programmers Interface).
Examples for application independent and collaboration-transparent systems are application sharing environments such as NetMeeting, VNC or Netviewer. They allow sharing the view of any single-user application among a group of users. Such systems only support strict WYSIWIS and are useful and effective for tightly coupled collaborative work, where independent interaction is not wished or not required. Multi-user free interaction where each user can individually for example work on a different part of a shared document is not supported.
An example for an application dependent system is the CoWord system by Xia, Sun et al. [10]. This systems makes use of the application specific API in order to integrate collaboration functionality into the Microsoft Word product. The user actions

performed on the word document are thereby intercepted and translated by the Adaption Layer into operations for the Operational Transformation (OT) Layer which is responsible for maintaining the consistency of the shared document. The collaboration system underlying CoWord is collaboration-transparent, supports relaxed WYSIWIS and can also be used in a collaboration-aware system design. For the collaboration-transparent approach CoWord requires the execution environment and the single-user application to provide a suitable API which can be used to intercept and replay user input events and whose data and operational model are adaptable to that of the underlying OT technique. If an application and the execution environment provide a suitable API the greatest effort lies in implementing the translation of the user actions into operations required by the OT Layer.

Other collaboration-transparent systems such as the one presented by He et al. [14] use a similar technique as in CoWord to enhance a single-user application with collaborative functionality. Low-level I/O events such as keyboard and mouse events are intercepted and translated into semantic commands. A so called Communicator collects high-level messages (such as CAD commands and model data) and low-level messages and transmits those over the network to the other collaborating sites. There they are translated into execution environment GUI commands or application specific API calls.

The Intelligent Collaboration Transparency (ICT) project [16] also uses a similar technique as the above systems to enhance single-user applications with collaboration functionality. In addition to the CoWord approach the aim is to support sharing of familiar heterogeneous single-user editors.

The main problem of all of these systems is the effort of translating user actions into application semantic commands. For each single-user application that is to be enhanced, a translation layer has to be implemented. Because this is a difficult and time consuming task, the second generation of the ICT project (ICT2) uses a difference algorithm instead [15]. This new approach has shortcomings in the performance and the applications can not be heterogeneous but must be semi-heterogeneous. That is their interfaces can differ but the writing style must be the same.

The Flexible JAMM [17] (Java Applets Made Multi-User) project uses a different approach. Single-user applications are enhanced by replacing selected single-user components of the shared application with multi-user versions. This approach requires the underlying execution environment to meet certain conditions such as capabilities for process migration, run-time component replacement, dynamic binding and user input events interception and replay. The common interface of applications extended by JAMM is Java Swing and Java Object Serialization (JOS). In contrast to the other mentioned systems, the JAMM system does not require the development of a translation layer in order to convert user actions into application semantic commands or API calls as long as an application is based on Swing and all application classes are serializable. Although the number of Java Swing based applications has increased in the last years, the number of single-user applications that fulfill the mentioned requirements is small.

The CEFX approach is based on a common interface: the XML Document Object Model (DOM). If a single-user application uses a DOM internally, a suitable application API is not required in order to integrate CEFX into an existing

application. Such single-user applications can be easily extended without the effort of implementing a translation layer. Another advantage is that heterogeneous applications can be used to collaboratively work on a shared document because they share a common interface, the DOM. The number of existing and emerging XML applications and thus the number of single-user applications using XML as data model (native XML applications) is growing. Today the majority of these applications are general XML editors or specialized SVG graphic editors[9]. Another example for an application that extensively uses the DOM is OpenOffice. OpenOffice uses OpenDocument[10], an XML format, as native file format. An increasing number of applications exist that do not necessarily use XML as their internal data model, but provide a DOM API that allows directly accessing and manipulating the internal data. These applications can also be extended by CEFX with relative small implementation effort.

In the case where an application does not base on XML and does not provide a DOM API, CEFX can also be used. This can be achieved by using similar methods as in the Transparent Adaption approach of CoWord or other transparent approaches. In this case it would be necessary to implement a DOM translation layer for converting user actions into XML operations for the CEFX collaboration layer. Technologies such as XML binding can dramatically reduce the time and effort of translating an internal data structure to the XML data model.

3 A Flexible Framework Architecture

Single-user applications that are extended with a collaborative framework are usually tied to its consistency model and conflict resolution strategy. Adjustments can become expensive and time consuming. In order to support as many different application types as possible, the consistency maintenance algorithm of a collaborative framework therefore has to be very flexible in the selection of conflict resolution strategies. Depending on the application type the required strategies may vary. For example, in one case a priority based conflict resolution strategy, where the user with the highest priority wins makes sense. In other cases a multi-versioning approach would be better. The CEFX framework allows an easy integration into existing applications (supporting the collaboration-aware and the collaboration-transparent approach) and an easy extension of the framework itself with new features or enhanced concurrency control algorithms. This flexibility is made possible by the plug-in architecture of CEFX. In the automotive industry it is required that all documents used in the design process are always accessible via a central site. This is supported by the hybrid software architecture of CEFX, discussed in the next section.

---

[9] For a list of native SVG editors refer to: http://www.w3.org/Graphics/SVG/SVG-Implementations

[10] OASIS Open Document Format for Office Applications (OpenDocument). OASIS Standard May 2005

3.1 A Hybrid Software Architecture

The Collaborative Editing Framework for XML is based on a hybrid-architecture. The hybrid architecture is a mixture of both the centralised and the replicated architectures. Each site holds a client and a server process as well as a partial or complete copy of the shared data resource. Additionally a server site exists holding both the shared data resource and a server process.
All operations are executed locally before they are sent to the other sites to be executed, just like in the replicated architecture approach. Additionally the operations are also sent to the server site; there they are executed as well. The remote client sites execute the received operations only if necessary; that is if they have an interest in the operation. Next to the better responsiveness as in replicated systems, the hybrid approach has the advantage of having always a central site that holds the current correct version of the document. If a new site joins the session, a copy of this version can be obtained easily. It also can be used for synchronisation issues. Users working on a very large document may not interfere with each other, but if they do at any point, the system will know if their local version is up to date or not. The system can then, if necessary, either obtain a new copy from the server site (if for example too many changes have been made) or simply execute the operations on the relevant document part. In that case it is not necessary to obtain a copy of the whole document but only a copy of the relevant part.

3.2 The DOM Adapter

The Document Object Model (DOM) API is a common interface to the XML model tree. Applications that allow editing of XML documents usually own a DOM in order to manipulate their data model. Implementations of DOM exist for nearly all modern programming languages and operating systems. When developing a framework that allows enhancing existing XML applications with collaborative functionality, it makes sense to use this common interface as an entry point to these applications. The connection between an application and the CEFX framework is the DOM Adapter (DA). In the case of a collaboration-aware integration of CEFX, it provides a DOM interface (by derivation) to the application that is exactly equivalent to the application's original DOM interface. Neither the application code manipulating the DOM nor the DOM object itself, need to be changed. The DA is programmatically provided with a reference to the original DOM and replaces it with a version generated by the DA. In the collaboration-transparent approach, the DA is integrated into the used translation layer (TL) (DOM/DOM TL if a DOM API is provided by the application, otherwise DOM/API TL) instead of the application. The generated DOM implements all methods of the original version and integrates control structures that forward modification events to the CEFX controller. These modification events are then analyzed by the CEFX controller and delegated to the corresponding handlers for synchronization and awareness issues. In a single-user application the document that is to be edited is stored for example on the local file system. When the user opens a document, the document is parsed and initialized by the application. The DA provides functions for parsing and initializing a document as well. It checks if the document is

currently open in a collaborative editing session. If it is not part of a session, a new session is created. If the document is already opened in a session, the DA carries out the corresponding actions to connect the application with that session. The CEFX framework provides functions to retrieve information on documents in open sessions and to connect to those sessions. The controller instance of the CEFX framework analyses changes made to the document of an application. It generates events and delegates these to the corresponding modules or plug-ins of the framework. CEFX provides a set of default implementations for concurrency control including conflict resolution strategies and awareness widgets. These default implementations can be extended, or replaced with completely new implementations. For this the CEFX framework defines a set of extension points. An extension point declares a contract (using XML and programming language interfaces) that extensions must conform to. Other plug-in implementations that want to connect to that extension point must implement that contract in their extension.

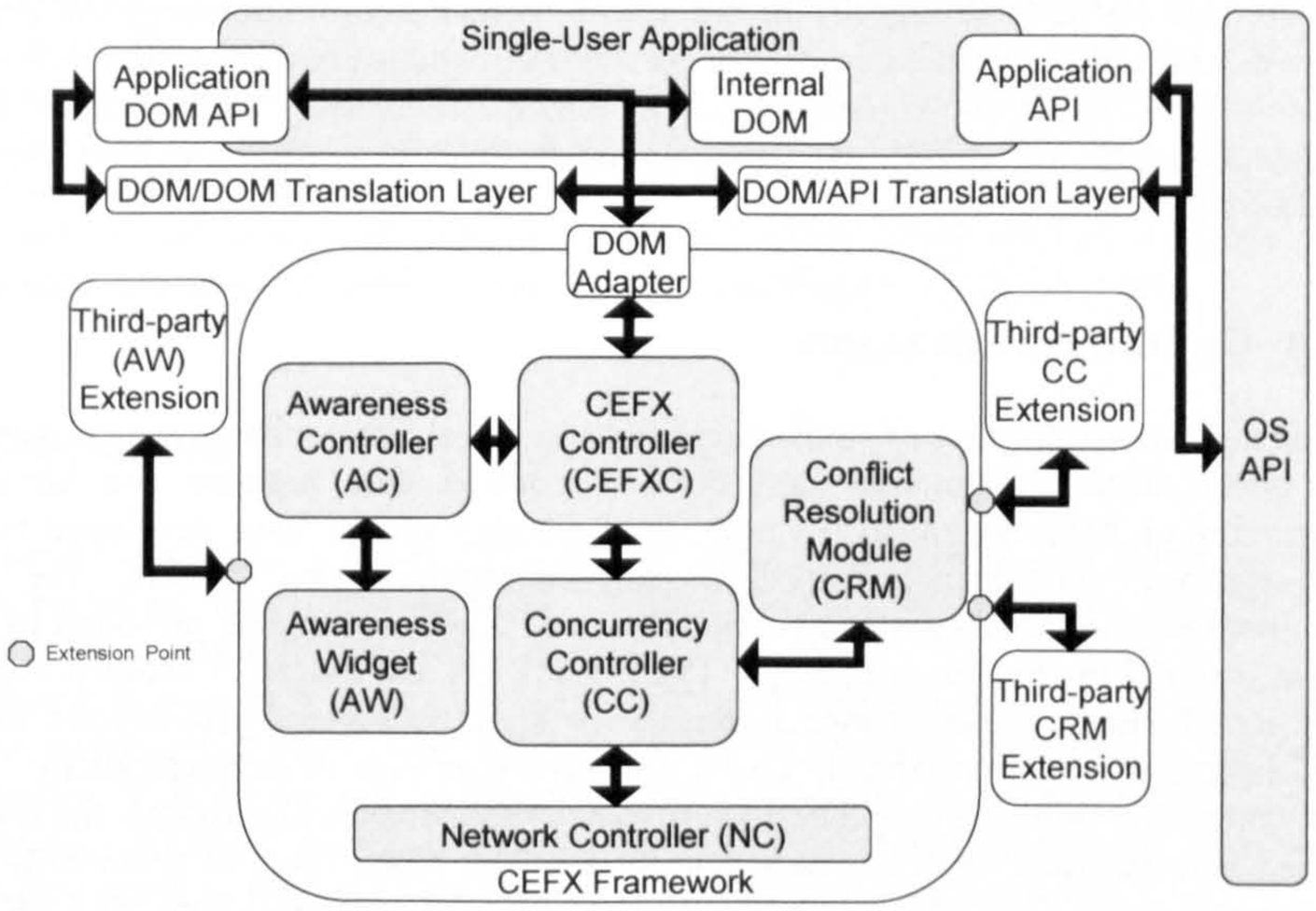

Fig. 1. The basic CEFX plug-in components and their interrelation. Plug-ins using extension points to extend the framework functionality are named third-party extensions in this figure.

The basic CEFX plug-in components, next to the DOM Adapter are the Concurrency Controller (CC), the Conflict Resolution Module (CRM), the Awareness Controller (AC), a set of Awareness Widgets (AW) and the Network Controller (NC). The CC of the framework is responsible for maintaining the consistency of the document and uses the NC module to transmit and receive editing events to and from remote sites. The CRM implements the rules that are applied if a conflict occurs. That is, how the system will react in that case. This could be for example on basis of a priority based or a no-operation conflict resolution scheme (CRS). A single-user application does not know the notion of remote sites. In order to give a user a feedback of what other

users are doing, so called awareness mechanisms need to be integrated into the application. Awareness mechanisms are also necessary to alert users of conflicts that can not be resolved automatically. For this integration, the framework offers methods to register own listeners and awareness widgets that get notified by the framework of events from remote sites. These listeners are registered with the AC. The AC is responsible for receiving and dispatching events from and to the application and remote sites. The events can for example contain information on mouse movements, locking events or other control events that need to be visualized to the user in some way or the other. Figure 1 shows an overview of the basic CEFX framework components and their interrelation. An application can use the default framework components such as awareness widgets or provide its own and register them with the framework using the extension points. The default awareness widgets themselves define extension points that allow extending them with new functionality. For functionality such as changing the conflict resolution strategy or selecting a part of a document that is to be locked, the framework offers control structures and extension points that can be used to configure the current concurrency controller and conflict resolution strategy implementations. The default concurrency controller and conflict resolution plug-ins define extension points as well in order to extend them with additional functionality.

3.3 Consistency Maintenance

For the synchronisation of XML documents in a real-time collaborative environment a consistency maintenance algorithm is required that supports the hierarchical structure of XML documents. A number of research groups have developed excellent consistency maintenance algorithms for hierarchical data models. The default implementation for consistency maintenance of XML documents provided by CEFX was inspired by the work of [2], [3], [5], [7], [8], [9] and others. A detailed discussion of our consistency maintenance algorithm for XML documents goes beyond the scope of this paper. Thus we can only give a very brief overview to its functionality.
In contrast to other, more general, consistency maintenance algorithms, the algorithm used here is based on XML as a data model. The finest level of granularity in this approach is an XML element node. Working on, for example character level, is not supported. If this is required an XML Schema could be used that defines a element node for each required granularity level, in this case for example character, word, sentence, paragraph, section, chapter and document. A character element node would contain only one character. A word element node would contain an ordered list of character nodes and so on. An important advantage of using XML as the data model is the hierarchical structure. Insert, delete and move operations can be performed on a higher semantic level such as a paragraph or a section. This increases the efficiency because there are fewer operations transmitted over the network and can help to enforce the semantic consistency of a document [2]. The basic set of operations used in the CEFX concurrency control algorithm is insert, delete, move and update. Additionally to support optional locking, lock and unlock operations are used. Each XML element node within a document is assigned a universal unique id (UUID). This allows addressing a node very quickly and unambiguously. This reduces the

processing time in comparison to algorithms that use a positional addressing scheme (e.g. [5]).

5 Requirements, Limitations and Future Work

If the collaboration-aware approach is used to enhance a single-user application, access to the source code of an application is required. Additionally the single-user application is required to use XML as internal data model. These requirements are met by a range of open source XML and SVG editing applications and some applications used in the automotive industry. If the collaboration-transparent approach is used, a translation layer (TL) has to be implemented that connects to the provided application API. If an application provides a DOM API the development of the TL is straightforward. Currently a prototype of the CEFX framework is being developed by the authors. The next step is to enhance an existing application with collaborative functionality using the CEFX framework. For this we will integrate our framework into an existing application for the design of electrical symbols, using the collaboration-aware approach. These electrical symbols are then stored in the CLS and used in the design of wiring diagrams. Future plans are to integrate this framework into an application for designing and editing of wiring diagrams.

6 Conclusions

This paper contributes a novel collaborative framework concept that allows an easy enhancement of single-user applications with real-time collaborative functionality. CEFX provides the following novel features:

- Flexible plug-in architecture allowing third party developers to extend CEFX with new awareness widgets, concurrency control mechanisms and conflict resolution schemes. This allows the framework to perfectly adapt to the requirements of the process workflow. Applications that use CEFX will profit from new third party developments as well, without the necessity of changing source code.
- Support for collaborative work using heterogeneous applications. This is especially helpful in the automotive industry, where different tools are used to manipulate, analyze and check the same data (in our case the vehicle electrical systems)
- Flexible consistency maintenance algorithm for XML documents supporting optional locking of document parts. This allows adapted working schemes and privacy support (not discussed in this paper)
- A novel approach to the extension of single-user applications by making use of the DOM as a common interface. The CEFX approach simplifies the integration of the collaborative framework into an existing single-user application supporting both, the collaborative-aware and the collaborative-transparent approach

Although the number of applications using XML as a data model today is relatively small, we assume that with XML becoming more popular the number of XML applications will increase. CEFX provides a way to easily enhance such applications

with collaborative functionality and we believe that its features promise a broad user-acceptance.

References

1. Chen, D. Sun, C, Jia, X. Zhang, Y. Yang, Y. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems. ACM Transactions on Computer-Human Interaction, Vol.5, No.1, March, 1998, pp.63-108
2. Ignat, C. Norrie, M. C. Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems. ACM. Fourth International Workshop on Collaborative Editing Systems, New Orleans, Louisiana. 2002
3. Molli, P. Skaf-Molli, H. Oster, G. Jourdain, S. Sams: Synchronous, asynchronous, multisynchronous environments. In Seventh International Conference on CSCW in Design, Rio de Janeiro, Brazil, 2002
4. Ellis, C.A. Gibbs, S.J. Concurrency Control in Groupware Systems. In Proceedings of the 1989 ACM SIGMOD international conference on management of data. ACM 1989
5. Davis, A. Sun, C. Lu, J: Generalizing Operational Transformation to the Standard General Markup Language. Proceedings of ACM 2002 Conference on Computer Supported Cooperative Work, Nov 16-20, New Orleans, Louisiana, USA.
6. Lamport, L. Time, clocks and the ordering of events in a distributed system. Mass. Computer Associates, 1978
7. Sun, C. Ellis, C. Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements. Proceedings of the 1998 ACM conference on Computer supported cooperative work , Seattle, Washington, 1998.
8. Galli, R. Data Consistency Methods for Collaborative 3D Editing. Ph.D. thesis, Universitat de les Illes Balears, Palma de Mallorca, Spain, Nov. 2000.
9. Ionescu, M. Marsic, I. An arbitration scheme for concurrency control in distributed groupware. In Proceedings of The Third International Workshop on Collaborative Editing Systems, pages 32–42. ACM, 2000
10. Xia, S. Sun, D. Sun, C. Chen, D, Shen, H.: Leveraging single-user applications for multi-user collaboration: the CoWord approach. Proceedings of ACM 2004 Conference *on Computer Supported Cooperative Work,* Nov 6-10, Chicago, IL USA
13. Wilkinson, N. Using CRC Cards: An Informal Approach to Object-Oriented Development. SIGS Books, New York, 1995.
14. He, F. Han, S. Wang, S. Sun, G. A road map on human-human interaction and fine-function collaboration in collaborative integrated design environments Computer Supported Cooperative Work in Design, 2004. Proceedings. The 8th International Conference on
15. Lu, J. Li, R. Li, D. A state difference based approach to sharing semi-heterogeneous single-user editors. CSCW'04 workshop on collaborative systems (IWCES-6) and application sharing systems. Nov. 2004, Chicago
16. Li, D. Li, R. Yu, Y. Yang, Y. Using Familiar Single-User Editors for Collaborative Editing. Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03)
17. Begole, J.M.A. Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems.[Ph.D. Dissertation. Virginia Polytechnic Institute and State University, Blacksburg, 1999

Erweiterung bestehender Anwendungen um kollaborative Funktionen mit Hilfe des Collaborative Editing Framework for XML (CEFX)

Ansgar Gerlicher
FH Stuttgart, Hochschule der Medien
Studiengang Medieninformatik
The London Institute, London College of Printing
gerlicher@hdm-stuttgart.de

Zusammenfassung: Das interdisziplinäre Forschungsgebiet des Computer Supported Collaborative Work (CSCW) befasst sich mit Gruppenarbeit und Zusammenarbeit, und den die Gruppenarbeit unterstützenden Informations- und Kommunikationstechnologien. Ein Teilbereich des CSCW ist das sogenannte „Real-Time Collaborative Editing". Dabei wird untersucht, wie Systeme aufgebaut sein müssen, die es mehreren Personen gleichzeitig erlauben, in Echtzeit auf einem Dokument zu arbeiten ohne dabei Inkonsistenzen zu riskieren. Die zur Zeit existierenden CSCW Editoren aktueller Forschungsarbeiten sind spezialisiert auf ein oder wenige Datenformate. Diese Forschungsarbeit hingegen beschäftigt sich mit der Entwicklung eines Frameworks (CEFX), mit dem es ermöglicht werden soll, quasi beliebige XML Dokument-Typen in Echtzeit gemeinschaftlich und über das Internet, also von beliebigen Standorten aus, zu editieren. Damit soll ein universeller Ansatz für die Lösung der Probleme des „Real-Time Collaborative Editing" geschaffen werden.

1 Einleitung

Die Erstellung größerer Dokumente wie zum Beispiel Kataloge, Handbücher, Software-Quellcode etc. erfordert oft Teamarbeit. Arbeiteten früher in einer Organisation mehrere Personen an einem größeren Dokument, so geschah dies meist nach dem einfachen Prinzip des „Turn-Taking". Dabei wurde das

Dokument von einer Person nach getaner Arbeit an die nächste weitergereicht. Auf diese Art und Weise war es möglich das Dokument konsistent zu halten und, da es nur eine jeweils gültige Dokumentversion gab, musste keine Synchronisierung oder Versionierung von Dokumenten stattfinden. Diese einfache Methode war allerdings sehr Zeitaufwändig und umständlich. Trotzdem wird heute oft noch so verfahren, gerade bei der Erstellung kleinerer Dokumente im Team, bei denen die Installation entsprechender Software zur Unterstützung der Teamarbeit zu aufwändig erscheint.

Um die Teamarbeit zu beschleunigen und zu vereinfachen und dennoch die Konsistenz der Dokumente zu wahren, werden heutzutage besonders im Bereich der Softwareentwicklung häufig Programme zur Versionierung und Synchronisierung von Dokumenten (in diesem Fall meist Quellcode) eingesetzt. Beispiele dafür sind Microsoft Visual SourceSafe [MSVSS] oder Concurrent Versioning System [CVS]. Diese Programme können im Prinzip für die Versionierung und Synchronisierung jedes Dokumenttyps eingesetzt werden. Für die Synchronisierung der Dokumentversionen werden dabei optimistische bzw. pessimistische Sperrverfahren eingesetzt. Beim pessimistischen Sperren ist paralleles Arbeiten in Echtzeit auf ein und derselben Datenquelle (dem Dokument) nicht möglich, da hier nur ein Anwender zu einer gegeben Zeit die Sperre besitzt. Erst nach der Freigabe des Dokuments durch diesen kann ein anderer darauf zugreifen. Beim optimistischen Verfahren ist paralleles Arbeiten zwar möglich, aber nur auf lokalen Kopien des Dokuments, wobei bei deren Zusammenführung zu einer neuen Dokumentversion Konflikte auftreten können. Paralleles Arbeiten direkt auf der Datenquelle ist bei keinem der beiden Ansätze möglich.

Ein Trend ist das kollaborative Arbeiten an verteilten Dokumenten im World Wide Web. Hierzu gibt es seit einiger Zeit Systeme, die Dokumente im Internet nicht nur lesbar, sondern auch editierbar machen. Hierzu zählen Technologien wie WikiWikis [WIKI] oder WebDAV [WDAV]. Bei diesen Systemen kommen dieselben oder ähnliche Verfahren zum Einsatz, wie bei den oben genannten.

Damit ist das parallele Arbeiten mehrerer Personen in Echtzeit auf derselben Datenquelle ebenso wenig möglich, da die Granularität der eingesetzten Sperrverfahren zu grob ist. Es ist meist nur möglich komplette Dokumentinstanzen zu sperren. Die hierarchische Struktur eines Dokumentes wird dabei nicht ausgenutzt.

In der Forschung existieren kollaborative Editoren [SUN1] für spezielle proprietäre Datenformate, die paralleles Arbeiten in Echtzeit mehrerer Personen über das Internet an einem Dokument unterstützen [SUN2]. Dazu zählen Systeme wie REDUCE [REDUCE], CoWord [CWORD] oder SAMS

[SAMS]. Diese Systeme ermöglichen allerdings nur das Editieren eines bestimmten Dokumenttyps und verwenden meist ein proprietäres Datenformat. Eine Lösung, die kollaboratives Editieren verschiedenster Dokumenttypen in Echtzeit über das Internet unterstützt, existiert noch nicht.

Viele der neuen und bestehenden Anwendungen für die Erstellung und Verwaltung von Dokumenten setzen auf XML als standardisiertes Datenformat. Dazu zählen z.B. verschiedene Office Anwendungen, Editoren für Vectorgrafik und Multimedia (SVG [SVG], SMIL [SMIL]) und Editoren für virtuelle Welten (X3D [X3D]). Dieser Trend, XML als standardisiertes Datenformat einzusetzen bringt viele Vorteile und kann für verteiltes Arbeiten genutzt werden.

Diese Forschungsarbeit beschäftigt sich mit der Entwicklung eines generischen Frameworks, mit dessen Hilfe Standardanwendungen, welche XML als Datenformat verwenden, mit der Fähigkeit zum kollaborativen Editieren von Dokumenten in Echtzeit und über das Internet erweitert werden können. Das Collaborative Editing Framework for XML (CEFX) verwendet dabei die besonderen Eigenschaften des XML Datenmodels für die Synchronisierung und Versionierung der Dokumente. Folgende Hauptfunktionen sollen in dieser Arbeit umgesetzt werden:

- Standortunabhängiges kollaboratives und synchrones Arbeiten an Dokumenten in Echtzeit unter Verwendung des Internets als Datenkanal
- Unterstützung aller aktuellen und zukünftigen XML Dokumenttypen
- Einfache Integration in bestehende Editoren und Tools und deren Erweiterung mit kollaborativen Funktionen
- Verbesserung der Usability durch sogenannte „Awareness"-Mechanismen
- Semantische Erweiterung der Konsistenzerhaltung durch Kontextprüfung auf Basis von XML Schema

2 Nebenläufigkeitskontrollverfahren in kollaborativen Editoren

Nebenläufigkeitskontrolle ist ein häufig eingesetztes Verfahren um Inkonsistenzen in Datenbanksystemen vorzubeugen. Frei nach der Definition von Bernstein et al. [31]versteht man unter Nebenläufigkeitskontrolle die Koordination der Aktionen verschiedener Prozesse, die gleichzeitig auf gemeinsame

Daten zugreifen und sich damit potentiell behindern. Nebenläufigkeitskontrollmechanismen werden verwendet um die Konsistenz eines gemeinsam genutzen Dokumentes zu bewahren. Dabei gibt es prinzipiell zwei unterschiedliche Verfahrensarten: Verfahren zur Konfliktvorbeugung und Verfahren zur Konfliktauflösung.

Zu den Verfahren der Konfliktvorbeugung zählen unter anderem die, in der Datenbankwelt weit verbreiteten Sperrverfahren zum Beispiel das pessimistischen Sperrverfahren. Die einfachste Form der pessimistischen Sperrverfahren ist das exklusive Sperren (exclusive locking) von Datensätzen. Da dieses Verfahren bei vielen parallelen Transaktionen zu einer schlechten Systemleistung führt, wurde das shared lock (read lock) bzw. das SX-Verfahren (shared-exclusive lock) eingeführt. Um die auftretende Deadlock[1] Problematik zu Lösen, wurden weitere pessimistische Sperrverfahren entwickelt, wie zum Beispiel das Zwei-Phasen-Sperrverfahren. Solche pessimistischen Sperrverfahren werden sehr häufig in Datenbanksystemen und in Versionsverwaltungssystemen, wie zum Beispiel Microsoft Visual Sourcesafe [MSVSS] eingesetzt.

Ein Verfahren der Konfliktauflösung ist im Gegensatz dazu das optimistische Sperrverfahren. Dieses basiert auf der Annahme, dass kollidierende Transaktionen relativ selten auftreten und ein präventives Sperren von Datensätzen unnötig hohen Aufwand und Leistungseinbußen nach sich ziehen würde. Im Gegensatz zu den pessimistischen Sperrverfahren bleibt hierbei der Ablauf einer Transaktion bis zu ihrem Abschluss unberührt. Damit ist ein quasi paralleles Arbeiten an den selben Datensätzen möglich. Erst am Ende einer Transaktion wird festgestellt, ob ein Konflikt mit einer anderen Transaktion aufgetreten ist. Verschiedene Client/Server Systeme, wie zum Beispiel das Concurrent Versioning System [CVS], verwenden dieses oder ähnliche optimistische Verfahren zur Datensynchronisation. Beim CVS System werden Änderungen an den Datensätzen auf Client Seite zunächst ohne Sperrung (locking) durchgeführt. Sobald die Änderungen abgeschlossen (und „committed“) wurden, werden die Datensätze auf dem Server gesperrt. Danach wird in der Validierungsphase geprüft, ob Konflikte mit anderen Transaktionen vorliegen. Mithilfe von Zeitstempeln wird geprüft, ob ein Datensatz zwischenzeitlich geändert wurde. Falls die Validierung fehlschlägt, wird die Transaktion zurückgefahren und wiederholt, oder der Client wird über den Konflikt in Kenntnis gesetzt. Der Vorteil dieses Verfahrens ist die kurze Sperrzeit während der Validierungsphase. Ein Nachteil ist die relativ hohe Wahrscheinlichkeit von Validierungsfehlern, so dass dieses Verfahren hauptsächlich in Bereichen eingesetzt wird, in denen es nur wenige „gleichzeitige“ Benutzer gibt. Diese Einschränkung ist allerdings wiederum sehr von der Sperrgranularität abhängig. Mithilfe einer entsprechend feinen Sperrgranularität wäre die

[1] Zur Begriffserklärung Deadlock siehe Anhang.

Wahrscheinlichkeit für Konflikte bzw. Validierungsfehler entsprechend geringer. Die Sperrgranularität bestimmt die kleinste Einheit eines Sperrvorganges. Systeme wie [MSVSS] und [CVS] aber auch andere existierende Systeme zur Unterstützung der Arbeit im Team wie zum Beispiel WikiWikis [WIKI] oder WebDAV [WDAV] verwenden nur eine Sperrgranularität auf Dokumentebene. Das heißt es wird immer das komplette Dokument gesperrt, auch wenn nur ein kleiner Teil bearbeitet wird.

Pessimistische Sperrverfahren scheinen nicht für Echtzeiteditoren geeignet, da durch die Sperrung auf Dokumentebene ein quasi paralleles Arbeiten nicht möglich ist. Auch optimistische Sperrverfahren scheinen ungeeignet, da bei zu grober Sperrgranularität und zu vielen „gleichzeitigen" Benutzern die Häufigkeit von Validierungsfehlern und der Aufwand der Zusammenführung asynchroner Dokumentversionen zu groß wird bzw. nicht mehr automatisiert erfolgen kann.

Ein weiterer wichtiger Grund, warum Sperrverfahren für Echtzeit-Editoren eher ungeeignet sind, ist der relativ große Zeitaufwand, den die Sperrung an sich beansprucht und die damit enstehende Verzögerung auf Seite des Client. Diese Verzögerung würde in einem Editor stark den Arbeitsfluss stören und ist daher nicht akzeptabel.

Für kollaborative Systeme, welche quasi Echtzeitanforderungen besitzen, wurden daher andere Synchronisationsverfahren entwickelt, wie zum Beispiel das Verfahren der Operational Transformation.

3 Operational Transformation

Operational Transformation (OT) ist ein Verfahren zur Konsistenzerhaltung durch Konfliktauflösung. Der größte Vorteil der Operational Transformation gegenüber Sperrverfahren ist, dass eine Operation sofort, das heißt ohne Verzögerung, durchgeführt wird. Der Benutzer muss also, im Gegensatz zu Sperrverfahren, nicht warten, bis eine Sperre auf dem zu bearbeitenden Datensatz durchgeführt wurde. Bei der Operational Transformation werden Operationen direkt auf einer lokalen Kopie des Dokuments durchgeführt und danach an die anderen Clients verteilt und dort erneut durchgeführt. Eine Operation kann dabei, wenn sie von einem Client empfangen wurde, vor ihrer Durchführung zunächst transformiert werden. Die Transformation hat dabei das Ziel, die Intention der Benutzer beizubehalten und die auf allen Clients vorliegenden Dokumentkopien zu konvergieren [8]. Es gibt verschiedenste Vorschläge für Operational Transformation Algorithmen für Dokumente, die auf einem linearen Datenformat basieren. Einige davon sind unter anderem

dOPT [6], adOPTed [16], GOT [9], GOTO [7], SOCT2 [17]. Im Gegensatz zu linearen Datenformaten gibt es zur Zeit nur einen Algorithmus für Operational Transformation, welcher auf einem hierarchischen Datenformat basiert: Der treeOPT [8] Algorithmus.

Die meisten dieser Algorithmen arbeiten nach dem selben Prinzip der Konsistenzerhaltung. Hier soll ein kurzer Überblick darüber gegeben werden. Der Operational Transformation Algorithmus wurde entwickelt um die Probleme der Divergenz, Intentionsverletzung und der Kausalitätsverletzung[2] zu beheben. Ein kollaboratives Editiersystem wird dabei als konsistent bezeichnet, wenn es immer den Erhalt von Konvergenz, Intention und Kausalität gewährleistet.

Zur Bewahrung der Kausalität wird ein Zeitstempel-Verfahren auf Basis der „Vector-Logical Clock" [15, 17] verwendet. Dies erlaubt es sicherzustellen, dass eine Operation A, die nach der Kausalordnung[3] „vor" einer Operation B liegt (A < B), auch vor dieser ausgeführt wird, unabhängig davon in welcher Reihenfolge die Operationen jeweils eintreffen.

Um die Konvergenz und die Intention einer Operation zu erhalten wird eine totale Ordnungsrelation [7, 8, 30, 36] zwischen den Operationen definiert. Die totale Ordnungsrelation definiert, welche Operationen in welcher Reihenfolge auf der jeweiligen lokalen Kopie eines Dokuments – welche im jeweiligen Client („Site") in einem kollaborativen Editier-System vorliegt – ausgeführt wird. Zusätzlich werden alle ausgeführten Operationen der jeweiligen „Site" in einem History-Buffer gespeichert. Basierend auf der totalen Ordnungsrelation und dem History-Buffer wird ein Undo/Do/Redo Schema definiert. Dabei werden beim Eintreffen einer neuen Operation zunächst alle Operationen im History-Buffer, welche aufgrund der totalen Ordnungsrelation abhängig von der neuen Operation sind (also ihr nachstehen), rückgängig gemacht (Undo). Damit wird der ursprüngliche Dokumentzustand wieder hergestellt. Danach wird die neue Operation ausgeführt (Do) und danach wieder alle Operationen, die zuvor rückgängig gemacht wurden (Redo). Zusätzlich wird jede Operation vor ihrer Ausführung transformiert um auf die Änderungen zu reagieren, die durch die anderen Operationen hervorgerufen wurden. Um dies zu verdeutlichen zeigt die Abbildung 4 ein Szenario eines Editiervorganges ohne Transformational Operation. Dabei arbeiten zwei Benutzer an einem gemeinsamen Dokument, welches den Text „efect" enthält. Der Text kann durch die Operation Ins(p,c) modifiziert werden. Durch diese Operation wird ein Buchstabe c an Position p im Text eingefügt. Es wird angenommen, das die Position des ersten Buchstaben im Text 1 (nicht 0) ist. Die Benutzer generieren die Operationen O1 = Ins(2,f) und O2 = Ins(6,s).

---

[2] Zur Begriffserklärung Divergenz, Intentionsverletzung, Kausalitätsverletzung siehe Anhang.

[3] Zur Begriffserklärung Kausalordnung siehe Anhang

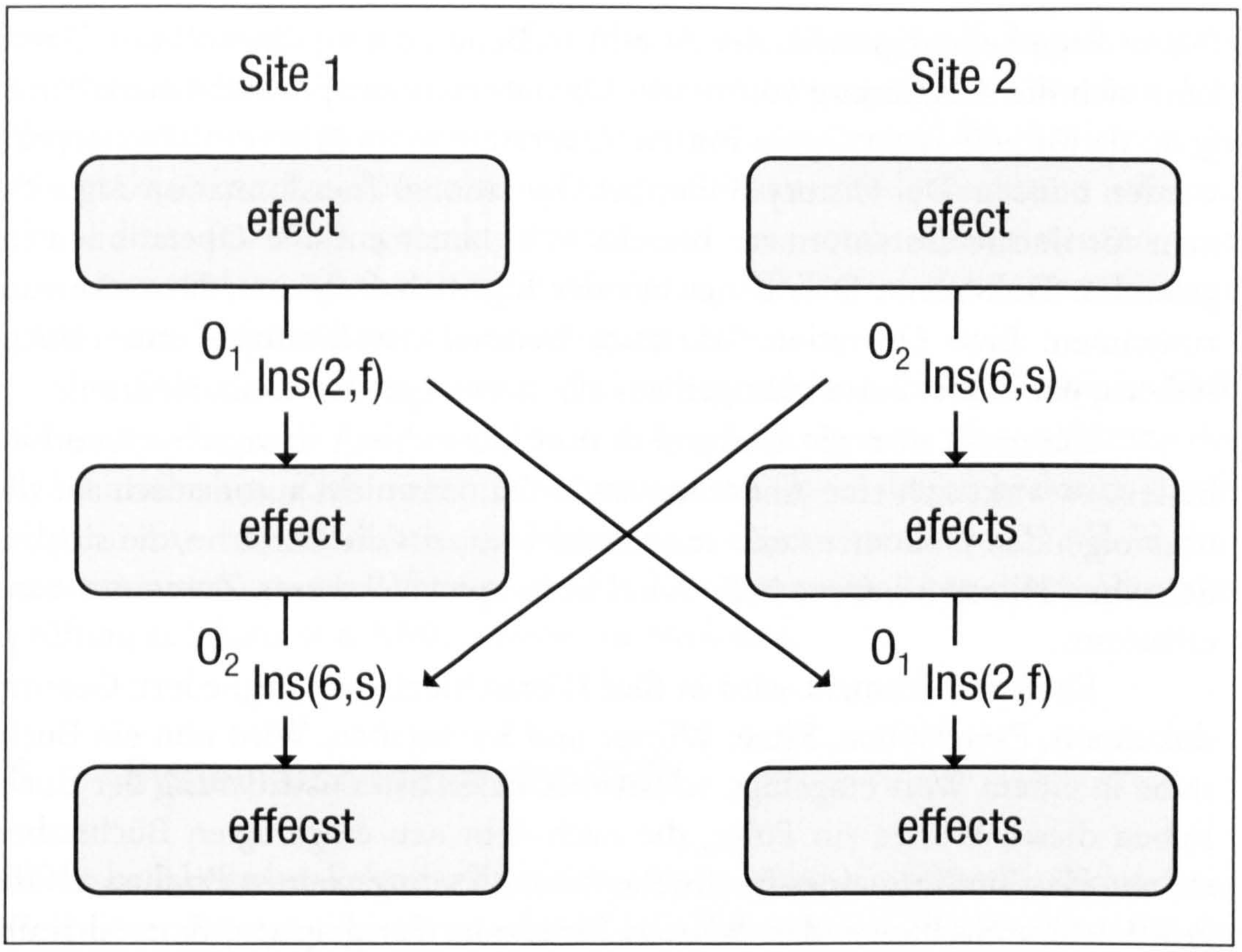

Abbildung 4: Fehlerhafte Integration von Operationen

Wird die Operation O1 an „Site" 2 empfangen und ausgeführt, so führt dies zu dem erwarteten Ergebnis „effects". Im Fall von „Site" 1 sieht dies anders aus. Dort wird nicht beachtet, dass die Operation O1 bereits zuvor ausgeführt wurde und damit die Textlänge verändert wurde. Das Resultat ist eine Divergenz der beiden lokalen Dokumente. Um ein korrektes Ergebnis zu erhalten, muss die Operation O2 unter Einbeziehung von O1 zuerst transformiert werden, bevor sie ausgeführt werden kann. Wird nun zum Beispiel die Operation O2 auf „Site" 1 in Ins(7,s) transformiert, so wird eine Konvergenz der Dokumente erzielt.

4 Synchronisation der XML Dokumente

Operational Transformation (OT) ist ein häufig eingesetztes Verfahren zur Synchronisation verteilter Dokumente in Echtzeit. Die meisten Algorithmen für Operational Transformation basieren allerdings auf einem linearen Datenformat. XML hingegen kann als hierarchisches Datenformat betrachtet werden. Bei der Operational Transformation kann ein hierarchisches Datenformat von Vorteil sein. Ein Problem der OT ist der oben erwähnte History-Buffer.

Dieser kann bei entsprechender Anzahl an Benutzern rapide wachsen. Damit kann sich die Ausführung von neuen Operationen entsprechend stark verzögern, da für jede neue Operation die Operationen im History-Buffer geprüft werden müssen. Der History-Buffer bei Operational Transformation Algorithmen für lineare Datenformate bezieht sich immer auf alle Operationen im gesamten Dokument. Dies hängt mit der Eigenschaft linearer Datenformate zusammen: Eine Operation, wie zum Beispiel das Einfügen eines Buchstabens, hat immer Auswirkungen auf alle nachfolgenden Dokumentteile.

Unterteilt man ein Dokument nun hierarchisch in verschiedene Einheiten, so wirkt sich eine Änderung am Dokument nicht automatisch auf alle nachfolgenden Dokumentteile aus, sondern nur auf die Bereiche, die sich auf derselben Hierarchieebene befinden. Ein Beispiel soll diesen Zusammenhang erläutern:

Ein Textdokument wird in fünf Hierarchieebenen gegliedert: Gesamtdokument, Paragraphen, Sätze, Wörter und Buchstaben. Wird nun ein Buchstabe in einem Wort eingefügt, so hat dies eine Positionsänderung der Buchstaben dieses Wortes zur Folge, die nach dem neu eingefügten Buchstaben stehen. Das Einfügen eines Buchstabens hat allerdings keinen Einfluss auf die Position der Nachfolgenden Wörter, Sätze oder Paragraphen, da es sich um ein hierarchisches Dokument handelt.

Dies hat für die Operational Transformation eine wichtige Bedeutung. Der History-Buffer muss nun nicht mehr für das Gesamtdokument gehalten werden, sondern für jede Hierarchieebene muss nun ein entsprechender Buffer angelegt werden. Dies scheint zwar Mehraufwand zu bedeuten, gleichzeitig bleiben aber die einzelnen History-Buffer viel kleiner und führen damit zu einem Vorteil in der Prüfung neuer Operationen und der Berechnung einer Operationstransformation. Die Prüfung einer neuen Operation muss gegen weniger Operationen im History-Buffer durchgeführt werden und auch die Berechnung der Transformation gestaltet sich einfacher.

Zur Zeit existiert nur ein Algorithmus für Operational Transformation, der auf einem hierarchischen Datenformat basiert [8]. Das dabei verwendete Datenformat ist proprietär und auf eine bestimmte Hierarchietiefe fixiert. Ziel dieses Forschungsprojektes ist es unter anderem, einen Algorithmus für Operational Transformation zu entwickeln, der eine variable und beliebige Hierarchietiefe erlaubt und als Datenformat den XML Standard unterstützt.

Operational Transformation ist sehr gut für die Synchronisation von strukturellen Operationen wie Einfügen und Löschen von Daten geeignet, besitzt aber Schwächen bei der Synchronisation von inhaltlichen Operationen, wie zum Beispiel das Ändern eines Attributs. Hier kann die Operational Transformation keine Konflikte wie die Intentionsverletzung verhindern. Für

diese Problematik muss ein Algorithmus für Operational Transformation durch zusätzliche Verfahren zur Konsistenzerhaltung ergänzt werden. Des Weiteren kann durch bestimmte so genannte „Awareness" Mechanismen die Aufmerksamkeit des Benutzers auf eventuell auftretende Probleme während des Editier-Prozesses gelenkt werden, um so vorab Inkonsistenzen und Intentionsverletzungen zu vermeiden. Das CEFX wird daher verschiedene „Awareness" Mechanismen zur Verfügung stellen.

XML bietet durch seine Eigenschaften, wie zum Beispiel eine in XML Schema vorliegende Grammatik weitere Möglichkeiten zur Unterstützung der Synchronisation. Ziel dieser Forschungsarbeit ist es unter anderem einen Algorithmus zu entwickeln, welcher diese Eigenschaften von XML Dokumenten nutzt um eine Verbessung der Konsistenzerhaltung durch Kontextprüfung auf Basis von XML Schema zu erreichen.

5 Systemarchitektur des CEFX

Viele kollaborative Echtzeit-Editoren verwenden die replizierte Architektur. Bei dieser Systemarchitektur gibt es im Gegensatz zu den meisten Client/Server Systemen, welche eine zentralisierte Architektur verwenden keine zentrale Datenquelle oder Server. Jeder Client besitzt dabei eine Kopie des Server Prozesses und der gemeinschaftlich genutzten Datenquellen. Der Vorteil einer replizierten Architektur besteht in den schnellen Antwortzeiten bei optimistischer Ausführung von Operationen. Wird eine lokale Operation generiert, so wird diese sofort ausgeführt und das Ergebnis wird damit sofort sichtbar. Im Gegensatz zur zentralisierten Architektur ist dabei kein Aufbau einer Datenverbindung zum Server oder gar eine sofortige Validierung der Operation (Stichwort: Sperrung) notwendig, was zu einem deutlichen Geschwindigkeitsvorteil führt. Die lokal ausgeführten Operationen werden danach an alle weiteren Clients übertragen und dort ausgeführt. Dabei kann es zu den schon besprochenen Inkonsistenzen kommen.

Die Systemarchitektur, auf der das CEFX basieren wird, ist eine Mischung aus der zentralisierten und der replizierten Systemarchitektur. Bei dieser so genannten Hybrid-Architektur gibt es einen zentralen Server, der immer eine aktuelle Dokumentversion besitzt. Die verschiedenen Clients besitzen jeweils lokale Kopien des gemeinsam genutzten Dokuments. Wie auch bei der replizierten Architektur, werden Operationen auf dem Dokument sofort lokal ausgeführt und danach an die anderen Clients übertragen. Dort werden die Operationen in dem entsprechenden History-Buffer gespeichert. Zusätzlich werden die Operationen auch an den Server übertragen. Da

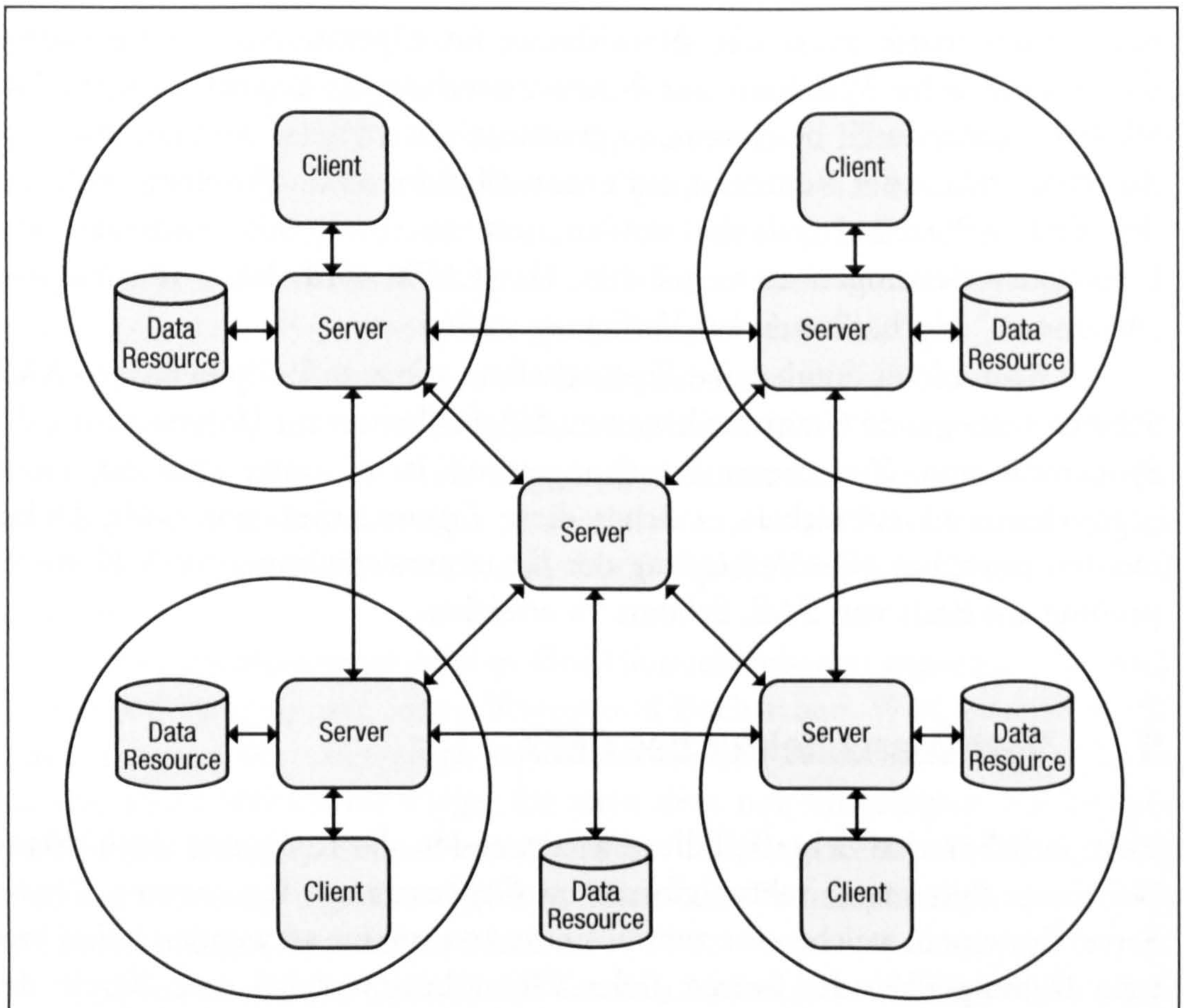

Abbildung 5: Hybride Systemarchitektur des CEFYX

das System auf XML als Datenformat basiert, müssen Operationen, die ein Client empfängt, nur bei Bedarf sofort ausgeführt werden: nämlich nur dann, wenn ein Benutzer den Bereich eines Dokumentes betrachtet, auf dem ein anderer gerade Operationen durchführt. Auf dem Server werden hingegen alle Operationen sofort ausgeführt. Somit liegt auf dem Server immer eine aktuelle Version des Dokuments vor. Dies ist neben den schnellen Antwortzeiten beim Editieren ein weiterer Vorteil der hybriden Systemarchitektur. Eine Kopie des aktuellen Dokuments kann dadurch zum Beispiel immer abgerufen werden, sobald ein neuer Benutzer am kollaborativen Editier-Prozess teilnehmen will. Auch kann das Dokument auf dem Server für die Synchronisation genutzt werden. Arbeiten mehrere Personen zum Beispiel an einem sehr großen Dokument, so ist die Wahrscheinlichkeit, dass diese sich eventuell gegenseitig stören relativ gering. Betrachtet nun ein Benutzer einen Teil des Dokuments das von einem anderen Benutzer intensiv bearbeitet wurde, so muss zuvor die lokale Kopie des Dokuments synchronisiert werden. Dies kann nun entweder durch Ausführen der Operationen im lokalen History-

Buffer geschehen, oder durch Abrufen der benötigten Teile der aktuellen Dokumentversion vom zentralen Server. Enthält der lokale History-Buffer zum Beispiel sehr viele Operationen, so ist es eventuell schneller, die aktuellen Teile vom Server nachzuladen, als die entsprechenden Operationen zu prüfen, zu transformieren und auszuführen.

Die beim CEFX verwendete Architektur macht erst durch die hierarchische Struktur des zu Grunde liegenden Datenformats (XML) Sinn. Bei Systemen, die auf einem linearen Datenformat basieren, würde diese Architektur eher hinderlich wirken. Dort hat sich die replizierte Systemarchitektur etabliert. Dieser neue Ansatz verspricht eine effiziente Möglichkeit zur Synchronisation verteilter Dokumente zu werden und macht damit ein Editieren in Echtzeit möglich.

6 Ausblick

Der Einsatz von XML in freien und kommerziellen Produkten aus verschiedensten Bereichen nimmt ständig zu. Eines von vielen Beispielen ist das OpenOffice Produkt von Sun Microsystems, welches als Speicherformat für Dokumente XML einsetzt. Auch Microsoft kann sich vor der XML-Welle nicht verschließen und bietet in der neuesten MS Office Version nun auch die Möglichkeit an, XML Dokumente zu importieren.

Das „Collaborative Editing Framework for XML“ (CEFX) soll eine einfache Möglichkeit schaffen beliebige Editoren, welche auf dem Datenformat XML basieren, mit kollaborativen Fähigkeiten auszustatten. Dazu wird eine Programmierschnittstelle angeboten, die eine einfache Einbindung in bestehende Editoren ermöglichen soll. Das so genannte Application Programming Interface (API) des CEFX soll dabei neben den normalen Funktionalitäten wie Einfügen, Löschen, Verschieben und Ändern von Elementen im vorliegenden XML Dokument auch „Awareness“ Mechanismen bereitstellen. Dabei sollen bestimmte Metainformationen zum Beispiel über den aktuellen Fokus eines Benutzers im Dokument über „Call-Back“ Funktionen abgerufen und im Editor entsprechend dargestellt werden können. Eine zusätzliche Validierungsfunktion der Benutzereingaben soll auf Basis von XML Schema eine quasi semantische Vorprüfung der durchgeführten Dokumentänderungen erlauben. Damit können inhaltliche Fehler im Voraus vermieden werden. Da das Framework eine generelle Unterstützung von XML als Datenformat vorsieht, soll praktische jede Form von XML Anwendung damit unterstützt werden können.

7 Anhang

Einige Begriffserläuterungen:

Divergenz

Die Abbildung 1 zeigt ein Szenario, bei dem drei Personen auf einem gemeinsamen Dokument in Echtzeit arbeiten. Dazu wird vor dem Editier-Vorgang vom Dokument jeweils eine lokale Kopie gemacht, auf der die Teilnehmer dann Änderungen durchführen. Die Abbildung 1 zeigt dazu den zeitlichen Ablauf der Operationen 01 bis 04, wie sie bei den verschiedenen Teilnehmern (Site 1-3) durchgeführt werden. Dabei werden die Operationen in jeweils unterschiedlichen Reihenfolgen ausgeführt. Die Reihenfolge der Operationen „Site 1" ist 01,02,04,03, „Site 2" 02,01,03,04 und „Site 3" 02,04,03,01.

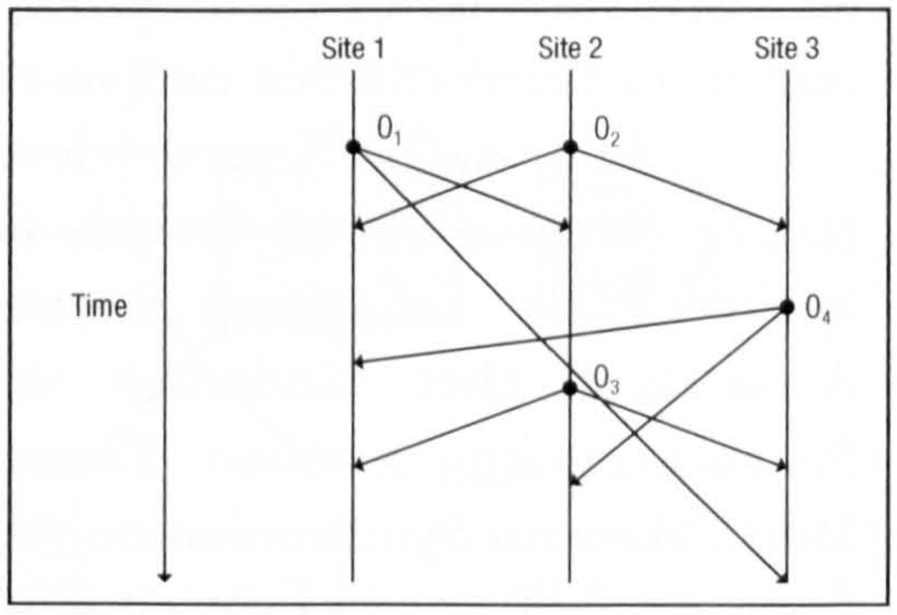

Abbildung 1:

Sind die Operationen 01,02,03,04 nicht kommutativ, so ist das endgültige Ergebnis der Operationen eventuell bei jedem Teilnehmer unterschiedlich. Eine Divergenz zwischen den verschiedenen lokalen Dokumenten liegt vor. Betrachten wir dazu die Baumstruktur eines einfachen XML Dokuments. Zu Beginn sind die beiden Bäume identisch.

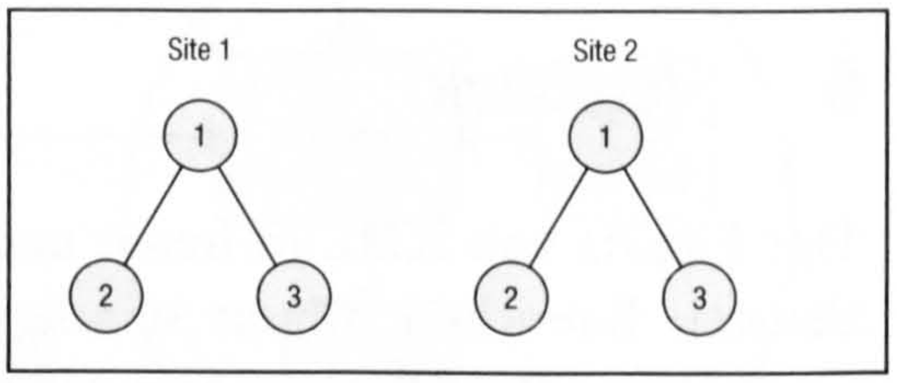

Abbildung 2.1: Beispiel Divergenz: zwei identische Bäume

Nun werden wie im obigen Szenario gezeigt verschiedene Operationen auf den Bäumen durchgeführt. Operation 01 erzeugt auf „Site" 1 einen Kindknoten A unterhalb Knoten 2 an erster Position. Operation 2 erzeugt auf „Site" 2 einen neuen Kindknoten B auch unterhalb Knoten 2 an Position 1. 01 und 02 werden quasi zeitgleich auf dem lokalen Dokument ausgeführt. Danach werden sie jeweils an den anderen Teilnehmer übertragen. Durch die Ver-zögerung in der Ausführung der Operationen, ist die Reihenfolge in welcher diese ausgeführt werden auf den jeweiligen lokalen Dokumenten unterschiedlich. Auf dem Dokument „Site 1" wird 01 vor 02 ausgeführt. Auf dem Dokument „Site 2" geschieht dies in umgekehrter Reihenfolge. Die beiden Dokumente sind divergent.

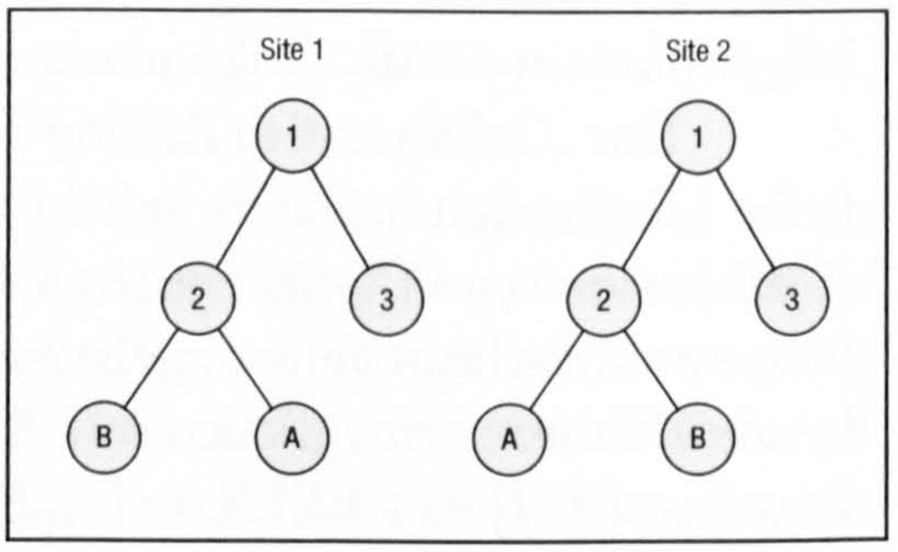

Abbildung 2.1: Beispiel Divergenz: zwei divergente Bäume

Intentionsverletzung

Eine Intentionsverletzung tritt dann auf, wenn zwei Operationen parallel zum Beispiel ein Attribut eines Objektes auf zwei unterschiedliche Werte abändern. Nehmen wir zum Beispiel einen kollaborativen Grafikeditor in dem die Operationen 01 und 02 parallel von zwei Benutzern ausgeführt werden. Operation 01 ändert dabei die Farbe eines Objektes innerhalb einer Grafik auf grün. Zur selben Zeit führt der zweite Benutzer Operation 02 auf dem selben Objekt in der selben Grafik aus, wobei 02 die Farbe des Objekts auf rot ändert. Nun werden die Operationen jeweils zum anderen Benutzer übertragen. Das Ergebnis ist, dass auf beiden Seiten das Objekt den falschen Farbwert besitzt. Es ist in diesem Fall nicht möglich den Konflikt ohne weiteres automatisch zu lösen, solange das betroffene Attribut nicht gleichzeitig mehrere Werte zulässt. In diesem Fall ist es nicht möglich einen konsistenten Eindruck dessen, was die Intention des jeweiligen Benutzers war, herzustellen. Eine Intentionsverletzung liegt vor.

Kausalitätsverletzung

Wie im Szenario in Abbildung 1 gezeigt, wird die Operation 03 erst nach der Ankunft von 01 bei „Site 2" generiert. Wurde 03 bei „Site 2" aufgrund von 01 generiert dann ist 03 von 01 abhängig. Zwischen 01 und 03 besteht eine so genannte kausale Abhängigkeit der Ereignisse. Auf „Site 3" wird nun im obigen Szenario die Operation 03 vor der Operation 01 ausgeführt. Ist 03 aber kausal von 01 abhängig, so entsteht bei „Site 3" eine Kausalitätsverletzung, da sich 03 auf einen zu diesem Zeitpunkt noch nicht vorhandenen Kontext bezieht. Eine Kausalitätsverletzung kann zu einem Zustand führen, in dem eine Operation nicht durchgeführt werden kann, da die Operation, von der sie abhängt noch nicht ausgeführt wurde.

Die Abbildung 3 zeigt, wie auf „Site 2" die Operationen 01 und 03 in der richtigen Reihenfolge ausgeführt werden. Operation 01 fügt dabei einen Knoten A unterhalb Knoten 1 ein. Operation 3 fügt danach einen Knoten B unterhalb Knoten A ein. Dies führt bei „Site 3" zu einem Zustand in dem Operation 03 nicht durchgeführt werden kann, da 01 noch nicht durchgeführt wurde.

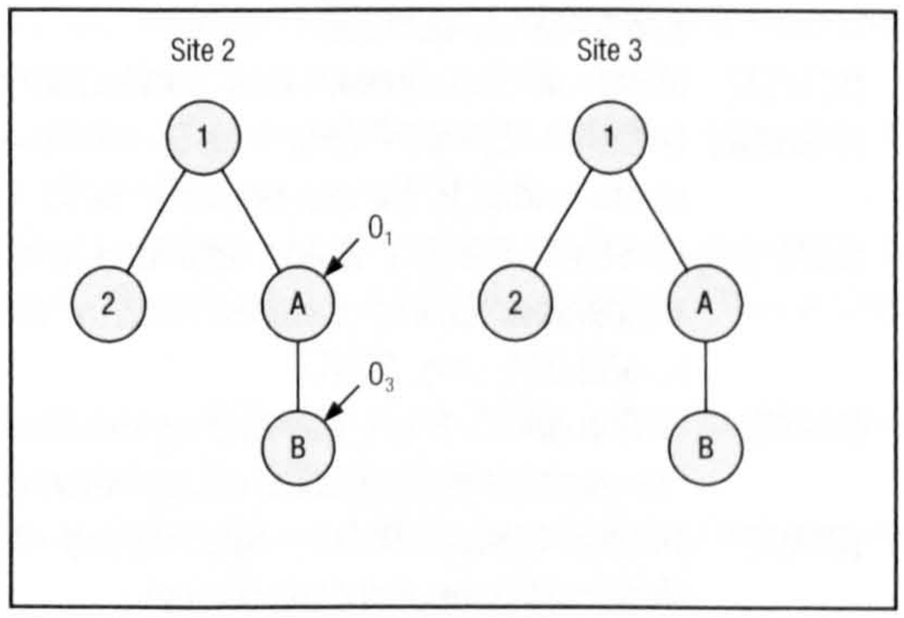

Abbildung 3: Kausalitätsverletzung

Kausalordnung

Die Kausalordnung ist eine Halbordnung, die über die Relation der kausalen Abhängigkeit über einer Menge von Ereignissen definiert wird: Ein Ereignis A ist eine Ursache von Ereignis B (A < B bzw. A liegt vor B) oder umgekehrt (A > B), oder die Ereignisse beeinflussen sich gegenseitig nicht (A || B), das heißt, sie sind kausal unabhängig oder nebenläufig. Die Kausalität wird zu dem von den meisten Theoretikern als transitiv betrachtet: Wenn Ereignis A eine Ursache von B ist, und B ist eine Ursache von C, dann ist A auch eine Ursache von C (wenn A < B und B < C ist, dann ist auch A < C). Andere wenden dagegen ein, dass zumindest unsere gewöhnliche Urteilspraxis bezüglich der Kausalität nicht transitiv ist, da wir bei der Suche nach der Ursache eines Ereignisses stets nach dem unmittelbar verursachenden Ereignis forschen.

Deadlock

Ein Deadlock (auch Verklemmung genannt) ist in der Informatik ein Zustand von Prozessen, bei dem mindestens zwei Prozesse untereinander auf Betriebsmittel warten, die dem jeweils anderen Prozess zugeteilt sind. Beispielsweise kann einem Prozess p1 der Bildschirm zugeteilt worden sein. Gleichzeitig benötigt p1 allerdings den Drucker. Auf der Gegenseite ist der Drucker dem Prozess p2 zugeteilt, der wiederum den Bildschirm fordert. Ein Beispiel für eine Verklemmung aus dem realen Leben ist eine Straßenkreuzung, an der von allen vier Seiten ein Auto gekommen ist und nun (die Regel rechts vor links vorausgesetzt) darauf wartet, dass das Auto rechts von ihm fährt. Nach Coffman et al. (1971) müssen vier notwendige Kriterien für einen Deadlock zutreffen:

1. Die Betriebsmittel werden ausschließlich durch die Prozesse freigegeben (No Preemption).
2. Die Prozesse fordern Betriebsmittel an, besitzen aber zugleich den Zugriff auf andere (Hold and Wait).
3. Der Zugriff auf die Betriebsmittel ist exklusiv (Mutual Exclusion).
4. Nicht weniger als zwei Prozesse warten in einem geschlossenen System (Circular Wait).

Deadlocks können bei Systemen eintreten, die fähig sind mehrere Prozesse parallel ablaufen zu lassen (Multitasksysteme) und bei denen die Reihenfolge der Betriebsmittelvergabe nicht festgelegt ist.

8 Referenzen

[CVS] *Concurrent Versions System. A open standard for version control.* http://ccvs.cvshome.org

[CWORD] C. Sun, Y. Yang, Y. Zhang, and D. Chen: *A consistency model and supporting schemes in real-time cooperative editing systems,* Proc. of the 19th Australian Computer Science Conference, Melbourne, S. 582-591, Jan. 1996.

[MSVSS] Microsoft Corporation. *Visual SourceSafe* http://msdn.microsoft.com/vstudio/previous/ssafe/

[REDUCE] D. Chen. *REDUCE – REal-time Distributed Unconstrained Collaborative Editing System.* Research Project at the Griffith University, Brisbane. 2001.

[SUN1] C. Sun, Y. Yang, Y. Zhang, and D. Chen: *A consistency model and supporting schemes in real-time cooperative editing systems,* Proc. of the 19th Australian Computer Science Conference, Melbourne, S. 582-591, Jan. 1996

[SUN2] C. Sun and D. Chen. *Consistency maintenance in real-time collaborative graphics editing systems.* ACM Transactions on Computer-Human Interaction. 2002.

[SAMS] H. Skaf, P. Molli. *SAMS – Synchronous Asynchronous Multisynchronous Editor.* Forschungsarbeit an der Université Henri Poincaré, Nancy1.

[SMIL] *SMIL – Synchronized Multimedia Integration Language.* www.w3.org/AudioVideo/

[SVG] *SVG – Scalabale Vector Graphics 1.0 Specification,* 2001: www.w3.org/TR/SVG/

[WDAV] *WebDAV. Web Distributed Authoring and Versioning.* Arbeitsgruppe der IETF. www.ics.uci.edu/~ejw/authoring/

[WIKI] Im World Wide Web verfügbare Seitensammlungen, die von den Benutzern nicht nur gelesen, sondern auch online geändert werden können. http://de.wikipedia.org/wiki/Wiki

[X3D] *X3D – EXtensible 3D.* Draft specification committed to ISO/IEC JTC1/SC24 for registration, December 2002: www.web3d.org/x3d.html

9 Literaturverzeichnis

Conference Proceedings:

[1] C. Sun and D. Chen. *Consistency maintenance in real-time collaborative graphics editing systems.* ACM Transactions on Computer-Human Interaction. 2002.

[2] D. Chen. *REDUCE – Real-time Distributed Unconstrained Collaborative Editing System.* Research Project at the Griffith University, Brisbane. 2001.

[3] A. H. Davis, C. Sun, and J. Lu. *Collaborative Editing of XML Documents. An Operational Transformation Approach.* International Conference on Supporting Group Work. ACM Proceedings. 2001.

[4] Peter Naur. *Revised Report on the Algorithmic Language ALGOL 60.* Communications of the ACM, Vol. 3 No. 5, S. 299-314. May 1960.

[5] R. Galli and Y. Luo. MU3D: *A Causal Consistency Protocol for a Collaborative VRML Editor.* ACM Proceedings. Web3D-VRML: Fifth symposium on Virtual Reality Modeling Language. 2000.

[6] C.A. Ellis and S.J. Gibbs. *Concurrency Control in Groupware Systems.* ACM Proceedings. SIGMOD International Conference on Management of Data. 1989.

[7] C. Sun and C. Ellis. *Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements.* ACM Proceedings. ACM conference on Computer supported cooperative work, Seattle, Washington. 1998.

[8] C. Ignat and M. Norrie. *Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems.* ACM. Fourth International Workshop on Collaborative Editing Systems, New Orleans, Louisiana. 2002.

[9] C. Sun, X. Jia, and et al. *Achieving Convergence, Causality-preservation, and Intention-preservation in Real-time Cooperative Editing Systems.* ACM. Transactions on Computer-Human Interaction. 1998.

[10] P. A. Franaszek, J. T. Robinson, and A. Thomasian. *Concurrecy Control for High Contention Environments.* ACM. Transactions on Database Systems. 1992.

[11] D. J. Rosenkrantz, R. Stearns, and P. Lewis. *System Level Concurrency Control for Distributed Database Systems.* ACM. Transactions on Database Systems. 1978.

[12] H. T. Kung and J. T. Robinson. *On Optimistic Methods for Concurrency Control.* ACM. Transactions on Database Systems. 1981.

[13] D. Gawlick. *Processing Hot Spots in High Performance Systems.* IEEE Springer. CompCon Conference. 1985.

[14] C. Ignat and M. Norrie. *Customizable Collaborative Editor Relying on treeOPT Algorithm.* Kluwer Academic Publishers. Eighth European Conference on Computer Supported Cooperative Work. 2003.

[15] C. J. Fidge. *Timestamps in message-passing systems that preserve the partial ordering.* University of Queensland, Australia. 11th Australian Computer Science Conference. 1988.

[16] M. Ressel, D. Nitsche-Ruhland, and et al. *An integrating, transformation-oriented approach to concurrency control and undo in group editors.* ACM. Conference on Computer Supported Cooperative Work. Nov. 1996.

[17] M. Suleiman, M. Cart, and et al. *Serialization of Concurrent Operations in a Distributed Collaborative Environment.* ACM. International Conference on Supporting Group Work. 1997.

[18] P. Dourish. *Extending Awareness Beyond Synchronous Collaboration.* Position paper. CHI 97 Workshop on Awareness in Collaborative Systems. 1997. S. 31.

[19] C. Gutwin, M. Roseman, and S. Greenberg. *A Usability Study of Awareness Widgets in a shared Workspace Groupware System.* ACM. Conference on Computer Human Interaction '96 Conference Companion. 1996.

[20] R. M. Baecker, D. Nastos, I. R. Posner, and K. L. Mawby. *The User-Centred Iterative Design of Collaborative Writing Software.* Conference on Computer Human Interaction '93. 1993.

[21] Abdessamad Imine, Pascal Molli, Gerald Oster, and Michael Rusinowitch. *Proving Correctness of Transformation Functions in Real-Time Groupware.* Kluwer Academic Publishers. ECSCW 2003: Eighth European Conference on Computer Supported Cooperative Work. Sept. 2003.

Books

[22] Brian W. Kernigham and Dennis M. Ritchie. *The C Programming Language,* Second Edition. Prentice Hall Inc. 1988.

[23] Th. Michel. *XML Kompakt. Eine praktische Einführung.* Page 36. Carls Hanser Munich. 1999.

[24] F. Arciniegas. *XML Developer's Guide.* Page 49. Franzis. Poing. 2001.

[25] F. Van der Vlist. *XML Schema.* O'Reilly. 2002.

[26] P. Walmsley. *Definitive XML Schema.* Prentice Hall PTR London. 2002.

[27] R. Wyke and A. Watt. *XML Schema Essentials.* Wiley New York. 2002.

[28] C.J. Date. *An Introduction To Database Systems.* Addison-Wesley. 2000.

[29] P. Bernstein, N. Goodman, and V. Hadzilacos. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley. 1987.

[30] L. Lamport. *Time, clocks and the ordering of events in a distributed system.* Computer Associates. 1978.

[31] P. Bernstein, A. V. Hadzilacos, and et al. *Concurrency control and recovery in database systems.* Addison-Wesley. 1987.

[32] T. Haerder. *Datenbanksysteme: Konzepte und Techniken der Implementierung.* 2. Edition. Springer. 2001.

[33] P. A. Bernstein. *Newcomer: Transaction Processing.* Morgan Kaufmann. 1997.

Articles

[34] P. Peinl. *Synchronisation in zentralisierten Datenbanksystemen.* Informatik-Fachberichte. 161. Springer. 1987.

[35] P. Butterworth, A. Otis, and J. Stein. *The GemStone Object Database Management System.* Communications of the ACM. S. 64-77. ACM. 1991.

[36] M. Raynal, M. Singhai, and. *Logical Time: Capturing Causality in Distributed Systems.* IEEE Computer. S. 49-56. IEEE. 1996.

Theses

[37] D. Chen. *Consistency Maintenance in Collaborative Graphics Editing Systems.* Dissertation at the Griffith University Brisbane. 2001.

[38] J. C. Lauwers. *Collaboration transparency in desktop teleconferencing environments.* Ph.D Thesis. Stanford, CA. 1990.

[39] Csaszar Lorant Zeno. *Real-Time Collaborative Graphical Editor.* Swiss Federal Institute of Technology Zurich. Institute of Information Systems, Global Information Systems Group. June 2003.

RFCs

[40] T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0.* RFC 1945. May 1996.

Links

[41] Microsoft Corporation. 2000. www.microsoft.com/windows/netmeeting/

[42] *The History of HTML.* www.w3.org/MarkUp/historicalW3C

[43] *Concurrent Versions System. A open standard for version control.* http://ccvs.cvshome.org