Department of Computer Science
University College London
University of London

Managing the Consistency of Distributed Documents

Christian Nentwich

Submitted for the degree of Doctor of Philosophy
at the University of London

November 2004

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Abstract

In many coordinated activities documents are produced and maintained in a distributed fashion, without the use of a single central repository or file store. The complex relationships between documents in such a setting can frequently become hard to identify and track.

This thesis addresses the problem of managing the consistency of a set of distributed and possibly heterogeneous documents. From the user perspective, the process of consistency management encompasses checking for consistency, obtaining diagnostic reports, and optionally taking action to remove inconsistency. We describe an approach to execute checks between documents and providing diagnostics, and demonstrate its validity in a number of practical case studies.

In order to be able to check the consistency of a set of distributed and heterogeneous documents we must provide mechanisms to bridge the heterogeneity gap, to organise documents, to find a way of expressing consistency constraints and to return appropriate diagnostic information as the result of a check. We specify a novel semantics for a first order logic constraint language that maps constraints to hyperlinks that connect inconsistent elements. We present an implementation of the semantics in a working check engine, and a supporting document management mechanism that can include heterogeneous data sources into a check. The implementation has been applied in a number of significant case studies and we discuss the results of these evaluations.

As part of our practical evaluation, we have also considered the problem of scalability. We provide an analysis of the scalability factors involved in consistency checking and demonstrate how each of these factors can play a crucial role in practice. This is followed by a discussion of a number of solutions for extending the centralised check engine and an analysis of their strengths and weaknesses, as well as their inherent complexities and architectural impact.

The thesis concludes with an outlook towards future work and a summary of our contributions.

Contents

1 Introduction **12**

1.1 Overview of Contributions 13

1.2 Statement of Originality 14

1.3 Thesis Outline 15

2 Motivation **16**

2.1 What is a Document? 16

2.2 Distribution and Consistency Checking as a Service 17

2.3 Document Heterogeneity 18

2.4 The Consistency Management Process 19

2.5 Related Work 20

2.5.1 Software Development Environments 21

2.5.2 Viewpoint-Oriented Software Engineering 23

2.5.3 Consistency Management in Software Engineering 23

2.5.4 Databases 25

2.6 Chapter Summary 26

3 Overview **27**

3.1 A Consistency Checking Service 27

3.2 XML as a Syntactic Representation 28

3.3 Introducing a Running Example 31

3.4 Chapter Summary 33

4 A Model for Documents and Constraints **34**

4.1 Basic Data Types 35
4.2 Document Object Model 35
4.3 XPath Abstract Syntax 38
4.4 XPath Semantics 39
4.5 Constraint Language 44
4.6 Chapter Summary 46

5 Checking Semantics **47**

5.1 Boolean Semantics 47
5.2 Link Generation Semantics 51
5.2.1 Consistency Links 52
5.2.2 Link Generation Semantics 54
5.2.3 On the Relationship of $\mathcal{L}$ and $\mathcal{B}$ 67
5.2.4 A Special Case 69
5.2.5 Some Properties 70
5.2.6 Discussion and Alternatives in the Semantic Definition 73
5.3 Chapter Summary 76

6 Implementation **77**

6.1 XML Constraints and Links 77
6.2 Document Management 79
6.3 Architecture and Deployment 81
6.4 Advanced Diagnosis 83
6.4.1 Using the Linkbase Directly 83
6.4.2 Report Generation 85
6.4.3 Link Folding 86

6.4.4 Tight Tool Integration 87
6.5 Chapter Summary 88

7 Case Studies **89**
7.1 Case Study: Managing a University Curriculum 89
7.2 Case Study: Software Engineering 91
7.2.1 Constraints in Software Engineering 92
7.2.2 Tool Support 95
7.2.3 Consistency in EJB Development 97
7.2.4 Evaluation 101

8 Scalability **106**
8.1 Scalability in Practice 106
8.2 Analysis 107
8.3 Incremental Checking 108
8.3.1 Background 109
8.3.2 Overview 110
8.3.3 Intersection 111
8.3.4 An Algorithm for Intersecting with Changes 114
8.3.5 Evaluation 118
8.3.6 Conclusions 121
8.4 Replication 122
8.4.1 Architecture 122
8.4.2 Evaluation 124
8.4.3 Conclusion 125
8.5 Distributed Checking 125

8.5.1 Requirements . . . 126
8.5.2 Architecture . . . 127
8.5.3 Design . . . 128
8.5.4 Implementation and Evaluation . . . 129
8.5.5 Conclusion . . . 132
8.6 Dealing With Very Large Documents . . . 132
8.7 Chapter Summary . . . 133

9 Related Work Revisited **135**

10 Future Work **138**

11 Conclusions **140**

A Wilbur's Bike Shop Documents **142**

B Constraint Language XML Syntax **145**

C Curriculum Case Study Rules **150**

D EJB Case Study Data **151**

D.1 UML Foundation/Core Constraints . . . 151
D.1.1 Association . . . 151
D.1.2 AssociationClass . . . 151
D.1.3 AssociationEnd . . . 151
D.1.4 BehavioralFeature . . . 151
D.1.5 Class . . . 151
D.1.6 Classifier . . . 152
D.1.7 Component . . . 152

D.1.8 Constraint 152

D.1.9 DataType 152

D.1.10 GeneralizableElement 152

D.1.11 Interface 152

D.1.12 Method 153

D.1.13 Namespace 153

D.1.14 StructuralFeature 153

D.1.15 Type 153

D.2 EJB Constraints 153

D.2.1 Design Model - External View Constraints 153

D.2.2 Design - Implementation Sample Checks 154

D.2.3 Design - Deployment Information Sample Checks 155

D.2.4 Implementation - Deployment Information Sample Checks 155

D.2.5 Implementation - Internal Checks 155

List of Figures

2.1 The Consistency Management Process 19

2.2 Consistency Management in Computer Science 21

3.1 Sample XLink 30

3.2 Wilbur's resources 32

4.1 Sample XML Document 35

5.1 Example Document for = and *same* 49

5.2 XML document for constraint examples 56

5.3 Definition of $\mathcal{L}$ for $\forall$ 57

5.4 Definition of $\mathcal{L}$ for $\exists$ 59

6.1 Constraint in XML 78

6.2 Sample linkbase in XML 78

6.3 Sample document set 79

6.4 Document set with JDBC fetcher 80

6.5 Relational table XML representation 80

6.6 Sample rule set 81

6.7 Checking component abstract architecture 82

6.8 Web checking service architecture 83

6.9 JMS message architecture 84

6.10 Dynamic linkbase view 84

6.11 Wilbur's Pulitzer report sheet 85

6.12 Report fragments for the running example 85

6.13 Generated report in a web browser 86

6.14 Out of line link in linkbase 86

6.15 Link inserted into document using linkbase processor 87

7.1 Sample shortened syllabus file in XML 90

7.2 Syllabus study timings 91

7.3 Automatically generated links in the curriculum 92

7.4 Curriculum inconsistency report screen shot 93

7.5 Constraints types with examples 94

7.6 Inconsistent UML model 96

7.7 EJB-Profile compliant UML design of a single bean 98

7.8 Consistency checks for EJB development 98

7.9 Example of a UML *standard* constraint 99

7.10 Example of an EJB profile *extension* constraint 100

7.11 Example of a deployment descriptor – implementation *integration* constraint 100

7.12 Example of a *custom* constraint 101

7.13 EJB consistency checks 102

7.14 Java structure representation in XML 103

7.15 Sample inconsistency - field from deployment descriptor not implemented . 104

8.1 Treediff output of changes to UML model 110

8.2 Illustration - subtree addition to document 113

8.3 Illustration - subtree deletion from document 113

8.4 Illustration - node value change 114

8.5 Number of rules selected for different change sets 119

8.6 Check time for different change sets 120

8.7 Replicated checker architecture 123

8.8 Document throughput vs. number of replicas 124

8.9 Distributed checker architecture overview 127

8.10 Distributed XPath evaluation 129

8.11 Newsfeed item fragment 130

8.12 Newsfeed rule 130

B.1 Constraint Language XML Schema Overview 145

List of Tables

5.1 Definition of $\mathcal{L}$ for *and* 60

5.2 Definition of $\mathcal{L}$ for *or* 62

5.3 Definition of $\mathcal{L}$ for *not* 63

5.4 Definition of $\mathcal{L}$ for *implies* 63

5.5 Definition of $\mathcal{L}$ for = ... 65

5.6 Definition of $\mathcal{L}$ for *same* 65

8.1 Document loading times in milliseconds 131

8.2 Check times in milliseconds 131

8.3 Distribution overhead 132

Acknowledgements

I am obviously indebted to my supervisors Wolfgang Emmerich and Anthony Finkelstein, whose sometimes diametrically opposite views on matters have certainly forced me to keep an open mind. The work contained in this thesis has been shaped by our shared appreciation of practical demonstrability, rapid prototyping, and use of open standards and previous research. We see these not just as engineering methods, but as valuable methods of conducting practical software engineering research, research that provides tangible increments to the state of the art and is readily transferable to practitioners.

I want to thank my examiners Perdita Stevens and Michael Huth, who had correctly identified flaws in the presentation, and argumentation of the original draft. I believe the present thesis is a much more mature, and well-rounded work as a direct result of addressing their comments.

I am fortunate to be witness and support to the ongoing commercialisation of our work in our company, Systemwire. I want to thank Ramin Dilmaghanian for his dedication to the company, as well as our initial users and countless number of academic institutions who have applied our products in related research. It does not happen frequently that one can see one's work so quickly transferred into practice.

The "xlinkit" project was always supported by a number of students, some of whom undertook large implementation projects to demonstrate the feasibility of various ideas. Carlos Perez-Arroyo single-handedly implemented the distributed checker, overcoming the rather harsh constraints I had imposed on not modifying the original check engine. He also provided the evaluation of the distributed checker. Suzy Moat implemented the secondary storage liaison mechanism for the check engine, and evaluated it against UML models of various sizes. And Licia Capra implemented the checker for the now superseded old constraint language and set up the first iteration of the curriculum case study. She also worked with Wolfgang and Anthony to specify the early document set and rule set mechanisms. Daniel Dui expressed the constraints for the Financial Products Markup Language, which I could then subsequently use as input for my repair action generation experiments. I am also grateful to, in no particular order, Zeeshawn Durrani for implementing the linkbase visualisation servlet; Clare Gryce, for expressing the UML Foundation/Core constraints for XMI 1.1; Anthony Ivetac, for making his Enterprise JavaBeans model and source code available for experimentation; Giulio Carlone for another linkbase servlet implementation; Aaron Cass from UMass for pointing out bugs in the check engine and suggesting improvements; Shimon Rura from Williams College, also for finding bugs and suggesting improvements; Fokrul Hussain for completing the work I had started on EJB checking by writing a more complete set of rules and reports; Tanvir Phull for demonstrating that it is possible to translate a subset of OCL into my constraint language; Ash Bihal for early

work on checking FpML; Nathan Ching, for writing a constraint deoptimizer that will be useful for many purposes in the future; Javier Morillo for working on an optimisation engine that can lead to potentially massive time savings on XPath evaluation; and finally, the 2nd year software engineering project group of 2000, for implementing the linkbase processor, XTooX, that was used in the curriculum case study.

Finally, I am grateful to Zühlke Engineering for supporting me financially. The UCL Graduate School has also contributed a generous scholarship that allowed me to focus on my work.

1 Introduction

Many businesses produce documents as part of their daily activities: software engineers produce requirements specifications, design models, source code, build scripts and more; business analysts produce glossaries, use cases, organisation charts, and domain ontology models; service providers and retailers produce catalogues, customer data, purchase orders, invoices and web pages.

What these examples have in common is that the content of documents is often semantically related: source code should be consistent with the design model, a domain ontology may refer to employees in an organisation chart, and invoices to customers should be consistent with stored customer data and purchase orders. As businesses grow and documents are added, it becomes difficult to manually track and check the increasingly complex relationships between documents. The problem is compounded by current trends towards distributed working, either over the Internet or over a global corporate network in large organisations. This adds complexity as related information is not only scattered over a number of documents, but the documents themselves are distributed across multiple physical locations.

This thesis addresses the problem of managing the consistency of distributed and possibly heterogeneous documents. "Documents" is used here as an abstract term, and does not necessarily refer to a human readable textual representation. We use the word to stand for a file or data source holding structured information, like a database table, or some source of semi-structured information, like a file of comma-separated values or a document represented in a hypertext markup language like XML [Bray et al., 2000]. Document heterogeneity comes into play when data with similar semantics is represented in different ways: for example, a design model may store a class as a rectangle in a diagram whereas a source code file will embed it as a textual string; and an invoice may contain an invoice identifier that is composed of a customer name and date, both of which may be recorded and managed separately.

Consistency management in this setting encompasses a number of steps. Firstly, checks must be executed in order to determine the consistency status of documents. Documents are inconsistent if their internal elements hold values that do not meet the properties expected in the application domain or if there are conflicts between the values of elements in multiple documents. The results of a consistency check have to be accumulated and reported back to the user. And finally, the user may choose to change the documents to bring them into a consistent state.

The current generation of tools and techniques is not always sufficiently equipped to deal with this problem. Consistency checking is mostly tightly integrated or hardcoded into

tools, leading to problems with extensibility with respect to new types of documents. Many tools do not support checks of distributed data, insisting instead on accumulating everything in a centralized repository. This may not always be possible, due to organisational or time constraints, and can represent excessive overhead if the only purpose of integration is to improve data consistency rather than deriving any additional benefit.

This thesis investigates the theoretical background and practical support necessary to support consistency management of distributed documents. It makes a number of contributions to the state of the art, and the overall approach is validated in significant case studies that provide evidence of its practicality and usefulness.

1.1 Overview of Contributions

This thesis makes a number of contributions to the state of the art. We provide a way to specify constraints that express relationships between elements in multiple documents, regardless of their storage location or their content. By building on an intermediate encoding and not incorporating location information these constraints take into account document heterogeneity and provide location transparency.

We specify a novel semantics [Nentwich et al., 2001b, Nentwich et al., 2001a] for the constraint language that maps formulae in the language to hyperlinks that highlight consistent or inconsistent relationships between documents. This powerful diagnostic is particularly effective in the distributed setting where hyperlinks are often used as a native navigation mechanism.

We move away from centralisation and demonstrate a lightweight architecture for managing the consistency of loosely maintained distributed documents [Nentwich et al., 2002]. This architecture is novel in that it raises consistency checking to the level of a first-class and standalone service, not a feature of any particular tool.

We include an assessment of the scalability of the proposed architecture, and an evaluation of a mixture of techniques such as incremental checking and distributed checking for addressing potential scalability limitations.

Finally, we demonstrate the applicability, and practicality of our work in substantial case studies [Nentwich et al., 2003b].

1.2 Statement of Originality

It is not always possible or productive to conduct research in isolation. During the course of producing this thesis I have had the fortune of having several students to support me in prototyping various ideas. They have been identified in the acknowledgements. In addition to this support, I have been able to build on foundations laid by others. This section serves to distinguish their contributions from the novel contributions I lay claim to in this thesis.

The approach to checking presented in this thesis has its roots in [Ellmer et al., 1999]. I was part of the research group that took on its continuation. This work specifically introduces the concept of relating XML documents using a constraint language, using *document universes* for structuring input and, crucially, the idea of connecting inconsistent elements using hyperlinks. It proposes a method of annotating the consistency rules with link generation strategies, for example "generate an inconsistent link if the rule is false".

The work presented in this thesis has adopted the technique of selecting elements and of generating hyperlinks for diagnostic purposes. It does however make significant advances that make the technique much more powerful and useable. In particular: the proposed language is based on first order logic and not a proprietary mechanism for specifying pairs of elements to be connected via hyperlinks. This has two important consequences: firstly, users now concentrate on specifying a constraint rather than expressing which elements are to be connected via a link – the link generation is *transparent* and handled by the link generation semantics. And secondly, it is possible to specify constraints, and thus generate links, between any number of elements rather than the source-destination pairs envisaged in the original paper. This raises the level of expressiveness considerably and enables the application of the system to complex documents such as those found in software engineering.

The *document set* and *rule set* mechanisms referred to in this thesis derive directly from the document universe mechanism. They have since been refined by Licia Capra in conjunction with Wolfgang Emmerich and Anthony Finkelstein to allow recursive set inclusion. I have further refined them to provide an abstraction mechanism that allows the inclusion of heterogeneous data sources such as databases or Java source files into a check. Anthony also produced the running example that is used throughout the thesis.

Danila Smolko has worked on consistency checking using mobile agents [Smolko, 2001]. His thesis includes an algorithm for analysing XPath expressions for the purpose of incremental checking. The algorithm attempts a simple static analysis of diff paths and rule paths, and failing that falls back to a slower evaluation method. The incremental checking techniques presented in this thesis proceed entirely on the level of static analysis, distinguishing the cases of element addition, deletion and change, and thus go significantly beyond Danila's earlier approach.

All other concepts and ideas in this thesis are entirely the result of my own research.

1.3 Thesis Outline

The content of the thesis is arranged as follows: The following two chapters will introduce the problem and neighbouring areas of research in more detail and outline, at a high level, our proposed approach. Chapter 2 explores the problem space and surveys related work, and Chapter 3 provides the overview.

Following the introductory chapters, we will move toward a formal definition of documents in Chapter 4 and use this to specify our novel semantics for creating links between inconsistent documents, in Chapter 5.

The thesis then explores the practical side of providing a consistency checking service, starting with the presentation of the implementation of our semantics in Chapter 6. Chapter 7 puts this implementation to the test in a number of case studies. In Chapter 8 we look at some of the limitations of our implementation, their impact on scalability, and possible solutions. This will complete the main body of the thesis – from the definition of the problem, a sketched high-level approach, a formal definition, practical implementation and substantial evaluation.

The remaining chapters revisit our survey of related work with some remarks in the light of the content of the thesis, provide a roadmap for future work, and state our conclusions.

2 Motivation

This thesis concerns itself with the problem of managing consistency between distributed, structured or semi-structured documents. It is our goal to find a solution that will help the user to track inconsistency, that will work in many application domains, and is amenable to straightforward implementation and evaluation.

In this chapter we will sketch the problem in more detail, define the scope of our work, including the types of documents we expect to manage, and look at the main research problems that have to be overcome in order to make progress towards a solution in the remainder of the thesis.

We will also embed the problem in the existing body of work on consistency management and related areas in Computer Science. This will help us to put our work into context, contrast it with previous work and further explain what is novel about our approach. A detailed comparison with related work will follow in Chapter 9 after the main discussion.

2.1 What is a Document?

"Document" is an overloaded term that is used differently by different people: a typical PC user will probably be thinking of a word processor document containing free form text; a web designer may be thinking of an XHTML [Group, 2002] file; and graphic designers may refer to their pixel graphics files as documents.

Clearly these types of documents are very different in nature, ranging from natural language, to markup, to graphical information. Dealing with inconsistency in natural language, between graphical information, and marked up or semi-structure data requires very different approaches, due to the difference in structural rigidity.

In our research we have focused on documents that contain *structured* or *semi-structured* data. That means that information in these documents is sufficiently grouped, or structured, to enable us to identify fragments of information and refer to them. This type of information encompasses the usual "text" forms of semi-structured, marked up documents like XML [Bray et al., 2000], database tables, source code files, and so on.

Another important characteristic of the types of documents we are interested in is that they are commonly used to store information that corresponds to some *model*, be it implicit or expressed formally:

- A Java source code file follows the Java grammar, as specified in BNF form.

- A product catalogue corresponds to the business model of what makes a product and a catalogue
- An XMI [OMG, 2000b] file usually conforms to the UML [Object Management Group, 1997] meta-model, or some other appropriate meta-model.

The clearly identifiable parts of these documents have something in common: their data items carry *static semantics* in the application domain. We want to check that these semantics actually hold.

2.2 Distribution and Consistency Checking as a Service

When related information is spread across a number of documents, the complexity of managing document relationships increases. In activities like software engineering, or business activities involving a large number of people, the number of documents can increase quickly, creating a consistency management problem.

Distribution occurs in two different ways in this scenario. Firstly, it refers to the scattering of information from a common, logical model across multiple documents. Secondly, it refers to the potential physical distribution of the documents themselves across multiple hosts.

Take the example of a UML model and a set of Java source code files. The documents are part of an overall model of a system, the UML model is a design model, and the Java source code an implementation model. The two are obviously related, and even if we do not wish to *enforce* consistency, executing a consistency check between them will help us track differences until we decide to take action.

Which tool is responsible for a consistency check across such heterogeneous specifications? Is it the UML tool or the Java source code manipulation tool, or both? Many modern development environments get around this particular problem by implementing support for both, and assuming responsibility for consistency, but what if we wish to add additional models later, for example our own proprietary information? What if we want to retain some openness to execute currently unforeseen types of consistency checks in the future? Where do we turn when we want to check our marked up requirements document, deployment or build scripts against the source code or UML model for traceability purposes?

We propose that it can be beneficial to treat consistency checking as a *service* in its own right; a service that sits between tools and executes consistency checks between heterogeneous documents. This makes clear the point of responsibility for checking, and removes the need to modify tools. Provided that the service can deal with heterogeneous notations, this also eliminates the need to integrate document models into a common

model and centralize them – a powerful means of ensuring consistency, but one that is not always appropriate due to the effort involved in achieving the integration, and the organisational politics involved in agreeing on such a model.

We also propose to distinguish the processes of checking consistency, reporting results and making changes to documents. This is an application of the "tolerant" approach to inconsistency, which has been proposed in previous research [Balzer, 1991] – we will have more to say on this area in a later section. A tolerant approach to inconsistency tracks and monitors inconsistency, but does not *enforce* its removal. By providing a checking service that tracks inconsistent relationships between documents we can keep the user informed and let them make their own judgment on when to take action, rather than forcing a particular process or way of working onto them.

The effect of adopting a tolerant approach is amplified in a distributed setting where documents may be owned by different people, and verbal communication may in any case be required before action can be taken. We believe that it is also a manifestation of a generally sound engineering principle: that of separation of concerns. If a stricter approach to inconsistency should be required, this could always be achieved by providing an editing, triggering and monitoring framework around a tolerant consistency checking mechanism, while keeping the two encapsulated in dedicated components. In a sense, this makes the tolerant approach more *general*, because it can be specialized with additional components to increase strictness.

In summary, we believe that in certain settings of distributed, heterogeneous documents, a standalone consistency checking service that supports a tolerant approach to dealing with inconsistency can provide tangible benefits, and will be useful as a light-weight solution to tracking complex relationships between documents.

2.3 Document Heterogeneity

Even under the assumptions set out previously, namely that we are dealing with structured or semi-structured documents that contain suitably accessible information, we must address the problem of document heterogeneity. Any approach that neglects this is liable to become domain specific or tool dependent.

Document heterogeneity must be addressed on two levels: *syntactic heterogeneity*, the physical representation of documents, for example as a text file or a database table, and *semantic heterogeneity*, which refers to the differences in the models we are trying to check against one another.

Syntactic heterogeneity can be addressed by falling back onto a common data representation. We will use open standards for this purpose and represent documents in XML. This

decision, the rationale behind it, and why we think that this does not lead to a loss of generality, will be discussed in Section 3.2.

Semantic heterogeneity is a harder challenge. Even if we represent documents in XML as a common physical format, we will still have to be able to relate different XML structures to one another. For example, a Java class may be represented either by its simple name, or fully qualified with its package name; information presented in a simple string in one document may have to be aggregated from multiple locations in another document, and so on.

Semantic heterogeneity can be addressed by either translating all input into a common model such as first order logic during the course of a check, or by creating a consistency checking mechanism that can directly relate information represented in different models. The former approach is probably more exhaustive and could be used to establish guarantees of completeness, but it also brings a large upfront cost that can make it infeasible if only certain parts of documents are ever checked for consistency. We opt instead for the second, by providing a constraint language that relates information in heterogeneous models. This makes our approach suitable for situations where a light-weight check of certain properties of documents is valuable and up-front costs need to be kept low.

2.4 The Consistency Management Process

We have already suggested that under our tolerant view of inconsistency, editing, consistency checking, and making changes to, or repair documents, are separable activities. These activities can be flexibly performed by the user, in a process that would resemble Figure 2.1.

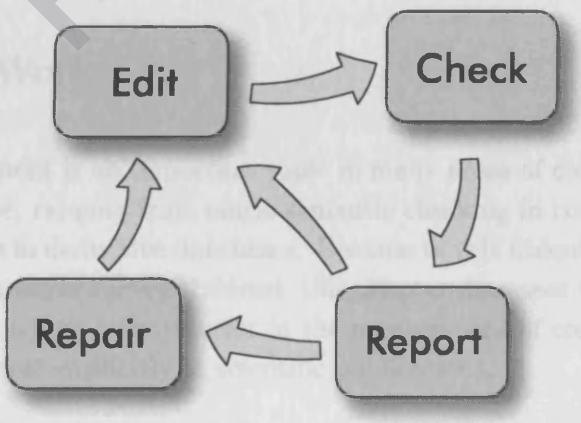

Figure 2.1: The Consistency Management Process

In this thesis, we will concentrate on providing components to assist with the checking

and reporting of inconsistency. As part of the consistency management process:

1. Users edit their documents. During this modification phase consistency is not considered an issue. At some point, for example during a baselining phase in software development, or a trade settlement run in a financial system, somebody initiates a consistency check.

2. The checker component provides a report, which contains diagnostics that express the current consistency status of the set of documents. Based on this consistency status, the user can make a choice to either repair the documents or to go back to editing.

3. If the user chooses to repair the documents, they use a tool to make the relevant changes. This may have to be preceded by face to face communication and perhaps even a conflict resolution process. We will not discuss these issues further in this thesis, but will instead concentrate on providing a solid checking infrastructure to support these activities.

Because we have clearly separated the different activities involved in consistency management, this process can be flexibly adapted to practical needs. The notion of tolerance towards inconsistency is highlighted by the absence of any mechanism that forces transition between the phases, or any sort of ordering. This does of course imply that in a real deployment some care must be taken that the results of a check reflect the current editing status of the document. We see this as the responsibility of a higher level workflow that could be built around these building blocks. In this thesis, however, we will mainly concern ourselves with providing the theoretical means and infrastructure for the building blocks themselves.

2.5 Related Work

Consistency management is an important topic in many areas of computer science. These areas are quite diverse, ranging from static semantic checking in compiler construction to integrity maintenance in deductive databases. Because of this ubiquity, it is not possible to even attempt an exhaustive survey. Instead, this chapter discusses those areas of research where there has been a long term interest in the management of consistency and where it has been addressed most explicitly in scientific publications.

It is not possible at this point to attempt a comparison of our approach to previous work in any amount of detail. Instead, we will concentrate on identifying the main research questions and the *weltanschauung* behind each area of related research. Chapter 9 will add depth to this summary through closer comparison.

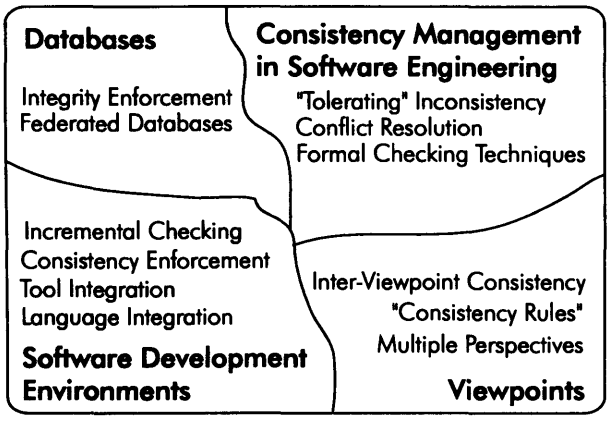

Figure 2.2: Consistency Management in Computer Science

Figure 2.2 is an overview to accompany the following discussion. It broadly classifies related research into four areas. This classification is a simplified guide. It is not exhaustive, nor are the areas mutually exclusive, for example the work on viewpoints and software development environments has frequently overlapped. For each area we discuss the most influential ideas. The following will be a systematic and largely historically ordered walkthrough of the field, beginning with software development environments.

2.5.1 Software Development Environments

Software development environments are the offspring of research on programming environments. The aim of a programming environment is to encapsulate the process of editing and compiling programming languages in a tool in order to increase feedback and limit the possibility of syntactic and static semantic mistakes. Inconsistency occurs when a user enters information that violates syntactic or semantic constraints.

Inconsistency is generally regarded as a "show stopper" in programming language compilation. Programming environments consequently evolved the concept of a "syntax-directed editor", an editor that would only allow the construction of legal source code according to a grammar. In this way, the possibility of making a mistake is eliminated altogether. From the early days, programming environments have thus included mechanisms for producing such editors automatically from language grammars.

The Cornell Synthesizer Generator [Reps and Teitelbaum, 1984] uses specifications based on ordered attribute grammars [Knuth, 1968] to automatically produce editors for programming languages. The editor maintains an attributed abstract syntax tree in memory, which is kept in a consistent state by checking user input against the attribute grammar. CENTAUR [Borras et al., 1988] uses a language called TYPOL [Despeyroux, 1988] for the specification of semantics and keeps multiple views of a program consistent by