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

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

List of Figures

List of Tables

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.

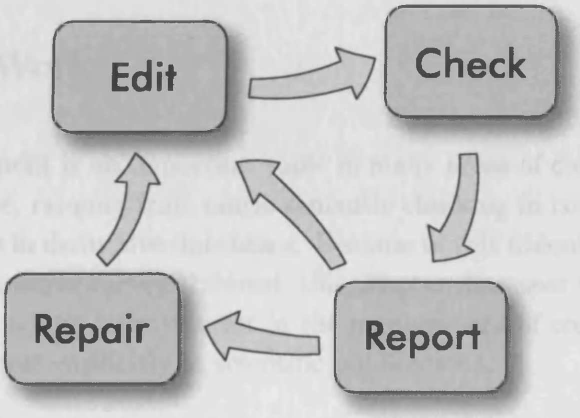


Figure 2.1: The Consistency Management Process

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.

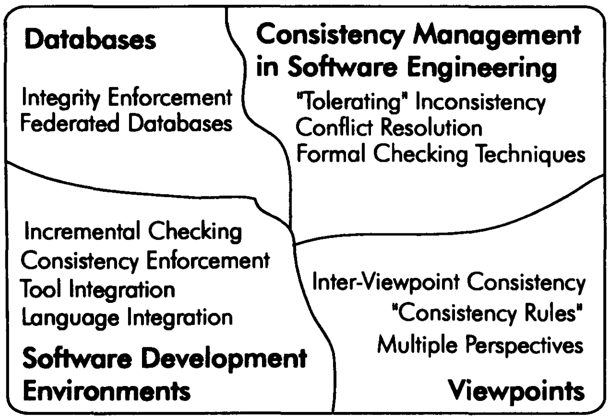


Figure 2.2: Consistency Management in Computer Science

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 checking if a change can be translated into a legal change on the abstract syntax tree. MultiView [Altmann et al., 1988] also uses abstract syntax trees, stored in a database, to provide multiple syntax-directed views of programs.

The idea of using abstract syntax trees for internal representation and providing static semantic checks was subsequently widened to include checks between heterogeneous languages. Software development environments can relate multiple abstract syntax trees using static semantic constraints and check if a construct in one language violates one in another. They also provide more comprehensive development support including features such as version control and project management.

Software development environments produced in research include the Eureka Software Factory [Schäfer and Weber, 1989], Gandalf [Habermann and Notkin, 1986] and GOODSTEP [GOODSTEP Team, 1994]. The latter allows the specification and checking of static semantic constraints between heterogeneous document types [Emmerich et al., 1995]. IPSEN [Nagl, 1996] also has some support for consistency checks between multiple views, but relies on the user to manually specify *references* between view elements.

These software development environments retain some of the characteristics of programming environments. In particular, it is assumed that abstract syntax trees for all documents are in memory, or are at least cached through an intelligent secondary storage mechanism. In practice this means that they tend to be built on a central repository, typically an object database. They also retain to some extent the view that inconsistency must be controlled at all times. GOODSTEP is a good example as it contains a built-in triggering system that provides a recheck after every editing operation.

Some modern, industrial successors of these early software development environments have moved away from such constraints in favour of a more tolerant approach to inconsistency. eclipse [IBM, 2003] is a good example, it checks for consistency of Java code bases on saving and provides a "task" view that warns of errors – but does not enforce resolution. This seems similar to the "design critic" feature in Argo/UML [Robins et al., 1999], which also maintains a user-modifiable list of consistency problems and more general issues. The consistency checking mechanism in both cases, however, still relies on a user workspace to contain all documents, and on in-memory abstract syntax tree representations being available at all times. It is thus not particularly extensible, as proprietary code has to be written into the tools to add support for additional types of documents.

In summary, research on programming environments concentrated on providing support for manipulating programs, with syntax directed editors that eliminate the possibility of introducing errors. Software development environments can be seen as an extension of this work to multiple heterogeneous document types, with additional features that support day to day software development. In both areas, we find reliance on a central repository, and mostly proprietary and not particularly extensible consistency checking mechanisms.

2.5.2 Viewpoint-Oriented Software Engineering

A Viewpoint [Finkelstein et al., 1992] is a specification fragment that describes part of a software system using a particular software engineering notation. Research on viewpoints emerged at about the same time as that on software development environments, but provides a looser approach to language integration.

The overall aim of Viewpoint-Oriented Software Engineering is to tackle the problem of coordinating the input of a large number of individuals in the design of large and complex software systems. The complexity of such systems typically leads to a requirement to specify them from different angles using different notations, for example to capture both the static and dynamic properties of a system. The problem of integrating such notations is known as the *multiple perspectives problem.*

It follows that inconsistency can arise in the viewpoints framework in two ways: a viewpoint can contain inconsistencies (*intra-viewpoint inconsistency*) or it can contain information that contradicts another viewpoint (*inter-viewpoint inconsistency*). It was envisaged that multiple viewpoints would be related using inter-viewpoint *consistency rules* [Nuseibeh et al., 1993, Easterbrook et al., 1994]. Alternatively, all viewpoints could be converted into a common first order logic representation [Finkelstein et al., 1994] and a theorem prover could be used to find contradictions in the combined logical specification.

The Viewpoints framework makes relatively few architectural assumptions about the distribution of documents. This is partly because it was never convincingly implemented or evaluated in practice. It did however take a step in the opposite direction of software development environments as far as its treatment of inconsistency was concerned. Inconsistency was not viewed as an error that had to be prevented, but as something that had to be detected and highlighted. It did not prescribe elimination but deferred the resolution of inconsistency to a later stage. This *tolerant* view of inconsistency has been influential and has since permeated into other areas of research that can be broadly classified as "consistency management in software engineering".

2.5.3 Consistency Management in Software Engineering

The tolerant treatment of inconsistency first introduced by [Balzer, 1991] was found to be a good approach to managing consistency in software engineering, where documents are complex and difficult to keep consistent at all times. Consistency management has since become a research topic of its own. The research agenda in [Finkelstein, 2000] shares many of our goals and provides a good overview of the problem space.

[Nuseibeh et al., 2000] discusses a high level framework for managing inconsistency in software development. This work is part of an ongoing interest in how to deal with inconsistency in software engineering artefacts [Nuseibeh and Russo, 1998, Nuseibeh et al., 2001]. It sets out the typical requirements, heterogeneity, the evolution of documents at different rates, and document complexity. An abstract architecture is specified that outlines the services necessary for consistency management: the location and identification of the cause of inconsistencies, the classification of inconsistencies, and actions for inconsistency handling. Work in this area has carefully addressed the organisational structures and processes necessary to support consistency management, as well as identified the high-level components that are needed to support it. Evaluation was however always based on manual methods that were geared towards specific specification languages. The work in this thesis can be seen to provide the concrete means to make these ideas feasible in practice.

[Zave and Jackson, 1993] propose to translate software engineering specifications into a common semantic model, represented in first order logic. Consistency checking between specifications can then be performed by taking the conjunction of their logical representation and detecting contradictions. The paper acknowledges that it will be difficult in practice to translate entire documents into logic, and even more difficult to prove properties about them. Instead, it is proposed that only properties that will lead to interaction between specification types are translated. It is not at all clear to us that it will be possible to easily identify such interacting properties as new types of documents are added.

Similar work in the area of viewpoints [Hunter and Nuseibeh, 1998] deals with the translation of specifications into first order logic, and the subsequent detection of inconsistency. A quasi-classical logic is used that enables the continuing analysis of specifications in the presence of inconsistency, circumventing the *ex falso quodlibet* property of classical logic. This idea, of ongoing reasoning in the presence of inconsistency, has proved popular in a number of application areas where formal analysis is required. [Easterbrook and Chechik, 2001] discusses a framework for model checking the composition of multiple, possibly inconsistent state models. Inconsistency is addressed by means of a lattice-based multi-valued logic, which enables the exploration of different merge scenarios, and subsequent resolution. [Huth and Pradhan, 2001] discuss further the problem of sound refinement of partial, view-based specifications, and how to reduce three-valued logics that can tolerate inconsistency in order to be able to apply standard model checking techniques. These approaches all share the common goal of allowing continuing analysis of, or reasoning with formal specifications in the presence of inconsistency. They benefit from the assumption that specifications are either already available as formal models, or can be translated into formal models. This makes it possible to provide powerful diagnostics such as action traces. In application domains such as safety critical systems, strong and proven diagnostics are imperative to enable developers to either eliminate, or control inconsistency. On the other hand, for simple documents translation into a formalism is perhaps more effort than is necessary, and being able to relate documents while retaining their native models may be desirable.

There are a number of neighbouring areas in software engineering that deal with model analysis, and the analysis of dynamic semantics, for example the work on model checking [Clarke et al., 2000]. In model checking, and in model analysis systems like Alloy [Jackson, 2002], a specification is taken and transformed by a proof system or trace algorithm into an internal model over which counter-examples can be generated. The generation of an internal state model usually implies resolving non-deterministic choices, which leads to the "state explosion problem". In Alloy, implied iteration over infinite models causes a similar problem, requiring the user to specify a maximum search horizon of instances. Inconsistencies in models are not reported back by pointing to a particular part of a specification, but by providing a trace, or an instantiated counter-example that shows how inconsistency arises. As we will see later, this differs from what we do in that we can point quite clearly to the document locations that cause inconsistency – because we have no notion of time, no implied state space, nor a notion of an implied model to worry about. Essentially, because we are attacking the smaller problem of static semantics, rather than dynamic semantics or anything that requires a proof system, we are able to provider firmer diagnostics.

2.5.4 Databases

Integrity has always been an important concept in database research, where the traditional view is that strong guarantees ought to be given on data consistency. In practice, this means eliminating inconsistency or preventing it from occurring in the first place through normalization of the data model. This view of eradication somewhat shifts the task of a consistency checker from detecting and pinpointing inconsistency to simply disallowing updates that would introduce inconsistency. A typical database system would rely on a mechanism such as ECA Triggers [Widom and Ceri, 1996] to recheck referential integrity and integrity constraints whenever a transaction modifies data, and issue a rollback or commit.

Federated databases [Ceri and Widom, 1993] are a means of making a collection of otherwise autonomous and centralized databases appear as a single entity. Research in the area has consequently aimed at extending the traditional referential integrity and transaction models from a single to multiple databases. The assumptions and design choices made in these systems – existence of transaction processing logic on each host, simple integrity constraints, enforcement of consistency – make them rather unsuitable for loosely managed distributed and heterogeneous documents. Later work [Grefen and Widom, 1997, Grefen and Widom, 1996] has successfully eliminated the need for heavyweight mechanisms such as global transactions, but still requires a certain amount of processing capability on each host. Furthermore, the assumption is that all data will be hosted in a relational database, which makes it difficult to include heterogeneous data into a check. We will discuss these questions further in Chapter 9.

2.6 Chapter Summary

In this chapter we have taken a more detailed look at the problem we are trying to solve. We will be working with structured or semi-structured documents that conform to some kind of model, or grammar. We will assume that documents may be heterogeneous on both a syntactic and semantic level, and that they may be distributed over a number of hosts on a network.

In order to retain domain independence, flexibility, and extensibility as well as supporting a tolerant approach to inconsistency, we have argued that consistency checking in this setting should be treated as a *service* that sits between tools and is invoked on request by a user.

We have then embarked on a tour of some of the research that forms the background of the thesis. We looked at the transition from programming environments that supported the editing of single document types to software development environments, which aimed to integrate a number of heterogeneous tools over proprietary, centralized data structures like abstract syntax trees or graphs.

The viewpoints framework is a loosely coupled system for capturing software engineering specifications in a number of heterogeneous languages. It features a notion of consistency checking and advocates a tolerant approach to inconsistency, in which consistency is monitored but not enforced. This interest in "lazy" consistency management has spun off a number of research projects in the area of software engineering, including the work on continued reasoning in the presence of inconsistency in formal models.

Finally, we have seen that databases have been successfully adapted to distributed scenarios using the federation approach. Research in this area initially relied on heavyweight mechanisms like global transactions and eventually moved to provide more lightweight communication protocols that allowed the construction of consistency monitors using local transactions only. The focus in this research was always to extend traditional database notions of integrity to allow a seamless transition to a distributed scenario.

With the necessary background established, we can now provide a high level overview of our approach to consistency management.

3 Overview

We have presented the problem of consistency management and introduced the notion of consistency checking as an open, tool-independent service. In this chapter, we will set the scene for the remainder of the thesis by sketching our approach for a service.

The following overview looks at the main characteristics of our service: the use of XML as a means of overcoming heterogeneity; the specification of constraints between multiple documents; and the reporting of inconsistency through a novel evaluation semantics that produces hyperlinks between inconsistent documents, and elements in the documents. Towards the end of the chapter, we will present a simple running example that will be referenced throughout.

3.1 A Consistency Checking Service

In the previous chapter we argued for the creation of a tool-independent, non-proprietary, open and extensible consistency checking service. This service should be able to deal with heterogeneous, structured and semi-structured documents, and monitor and track inconsistency between them.

We envisage that such a service could operate as follows:

- We retrieve documents and load them into memory, translating them into an XML representation in the process. This translation, which is discussed further in Section 3.2, acts to remove syntactic heterogeneity barriers.
- A constraint language, based on first order logic but adapted to work with XML, is used to establish constraints between multiple documents. This defines a simple notion of consistency – documents that obey constraints are consistent, and those that do not are inconsistent.
- We provide an evaluation semantics for the constraint language that effectively pinpoints where in documents constraint violations occur.

In order to effectively identify and report inconsistency we have defined a semantics for our constraint language that creates hyperlinks called *consistency links* between elements in documents –we will see later how documents are structured and elements in documents are identified. A consistency link is an n-ary hyperlink that carries a status attribute. The status attribute indicates whether the linked elements are in a consistent or inconsistent relationship. For example, if a constraint specifies that element a in document A has to be equal to element b in document B, we will link a to b using an "inconsistent" link in case of a violation.

We will see later that these links act as "witnesses" to constraint violations, recording the locations of inconsistent elements and persisting them in a set of links. Such a set of links acts as a snapshot of the state of consistency of a set of documents with respect to constraints. Hyperlinks have long been used as an effective means of navigating between distributed, semi-structured information and we use them to establish a diagnostic mechanism that is well suited to our problem.

We will also explore several ways to make use of the links to report consistency violations to the end user: by visualizing the links in a web browser; by providing textual reports that display data from the documents; or by inserting the links into the checked documents to enable direct navigation between inconsistent elements.

We have fully implemented our service and evaluated it in several case studies that evidence the usefulness and diagnostic power of our consistency link semantics. The implementation is contained in an encapsulated component and is largely architecture independent, making it deployable as a simple tool, or as an integrated service in an enterprise architecture.

3.2 XML as a Syntactic Representation

The checking service will have to be able to reference and manipulate elements in different types of documents in order to address heterogeneity. We use the eXtensible Markup Language (XML) [Bray et al., 2000] as an underlying syntactic representation. While we could use other data models to the same effect, XML provides a number of advantages.

XML has become very popular for data interchange. It is an open standard and has been rapidly adopted and implemented by the software industry. Users can create their own markup elements and attributes. The relationships of the markup elements, such as which elements can be contained in others, can be controlled by providing a grammar, in the form of a Document Type Definition (DTD) or XML Schema [Fallside, 2001]. XML files can then be checked, or *validated*, against the grammar to protect processing applications from syntax errors.

XML has simplified the creation of markup languages and this has led to its adoption in a variety of application domains. For example, in Software Engineering, XMI [OMG, 2000b] is a standard encoding for Meta Object Facility [Object Management Group, 2000a] instances, including the UML [OMG, 2000a]; business to business messaging is covered by ebXML [Eisenberg and Nickull, 2001]; and financial institutions are working on a standard format for derivative trade data, the Financial Products Markup Language [Gurden, 2001].

Most importantly for us, XML comes with a set of additional technologies that facilitate the processing of XML files. We will briefly take a look at the technologies we are using, the Document Object Model (DOM) [Apparao et al., 1998], XPath [Clark and DeRose, 1999] and XLink [DeRose et al., 2001].

The Document Object Model (DOM) is an application programming interface (API). It specifies a set of interfaces that can be used to manipulate XML content. XML content is represented in the DOM as an abstract syntax tree structure. The interfaces contain methods for manipulating nodes in the tree, traversal and event handling. The DOM provides a convenient mechanism for representing XML documents in memory and is implemented by most major XML parsers and XML databases.

Since the initial specification of XML, several languages have emerged as "core" languages that provide additional hypertext infrastructure to applications that have to deal with XML. XPath is one of these core languages. It permits the selection of elements from XML documents by specifying a tree path in the documents. For example, the path `/Catalogue/Product` would select all `Product` elements contained in any `Catalogue` elements in an XML file. The language includes mechanisms for traversing the tree along different *axes*. There are axes for navigating to the children of a node, as in the example, the parent node, ancestor nodes, sibling nodes and various XML namespace related axes.

XPath location paths can be absolute, like `/Catalogue/Product`, relative to a *context node*, like `Product` or relative to a *variable*, for example `$x/Product`. Relative paths have to provide an execution context that contains either a context node or provides variable resolution. XPath also permits the restriction of nodes along each step of the path according to *predicates*, boolean expressions that must be *true* if a node is to be selected. For example, the path `/Catalogue/Product[@name='X']` would select only those `Product` elements whose attribute `name` is equal to the literal string `'X'`. In practice, XPath processors like Jaxen [McWhirter, 2004] or Xalan [Apache Software Foundation, 2004] often return DOM constructs as a result, for example `Nodelist` objects that represent a node set. We will use XPath in this way throughout the thesis.

XLink is a another core infrastructure language. It is the standard linking language for XML and provides additional linking functionality for web resources. HTML links are highly constrained, notably: they are unidirectional and point-to-point; have a limited range of behaviours; link only at the level of files unless an explicit target is inserted in the destination resource; and, most significantly, are embedded within the resource, leading to maintenance difficulties.

XLink addresses these problems allowing any XML element to act as a link, enabling the user to specify complex link structures and traversal behaviours and to add metadata to links. Fig. 3.1 shows some XML markup that uses XLink to turn an element into a link. The `Product` element has an `xlink:type` attribute attached to it – XLink aware processors will now recognise this element as a link. The element contains two elements of type locator, which will be recognised as link endpoints. This link now links together the first and second Product element in oldcatalogue.xml. Most importantly, it links together two elements in a file without inserting any links directly into the file, that is it is an external link with respect to those files.

```
<Catalogue>
    <Product xlink:type="extended">
        <Name>Haro Shredder</Name>
        <Code>B001</Code>
        <Price currency="GBP">349.95</Price>
        <Combines xlink:type="locator"
                  xlink:href="oldcatalogue.xml#/Catalogue/Product[1]"
                  xlink:label="component 1"/>
        <Combines xlink:type="locator"
                  xlink:href="oldcatalogue.xml#/Catalogue/Product[2]"
                  xlink:label="component 2"/>
    </Product>
</Catalogue>
```

Figure 3.1: Sample XLink

Since such "extended links" can be managed separately from the resources they link, it is possible to compile "linkbases", XML files that contain a collection of XLinks. Linkbases can then be selectively applied to establish hyperlinks between resources. The XLink language contains further constructs for specifying behaviour during link traversal and traversal restriction, which this thesis makes no use of.

By representing all documents as DOM trees during a check and using XPath to reference nodes in the trees we can specify constraints between any type of structured or semi-structured document, regardless of its native representation. All that is required is a simple translation step before the check. We will also be able to point to inconsistent elements in documents using XLink. XML and its related technologies thus enable us to choose an effective, open and standardised approach to overcoming syntactic heterogeneity.

No Loss of Generality Whenever a particular technology is chosen as a key enabling factor, it is important to step back and analyse potential risks and the impact of future change. Technology evolves at a rapid pace and eventually XML will be superseded.

It will become clear that the approach proposed here is not at all dependent on XML, XPath and the other related technologies. Any structured representation could be employed instead, as long as it provides the following mechanisms:

- A sufficiently powerful grammar to be able to describe a wide variety of document types.

- An addressing mechanism that can be used to point to and select elements from documents – replacing XPath.
- A notion of a *unique identifier* for each element in the document.

As long as we can select elements from documents, we will be able to express constraints between them. We will also be able to provide targets for changes generated by a repair action generator. The use of XML as a particular implementation mechanism is thus not a limiting assumption. *Without loss of generality*, the languages and semantic specifications in this thesis will therefore be expressed by making direct use of mechanisms provided by XML. This will simplify the discussion and eliminate the need to specify formalisms on an abstract level that subsequently has to be mapped to an implementation.

3.3 Introducing a Running Example

In the following, we will frequently have to give examples of consistency constraints, particularly in the definition of our formal semantics. We introduce a running example here to use on such occasions, so that we have sample documents and constraints to refer to. This section also serves to provide a simple example of how inter-document consistency relationships arise, before we move on to more complex real world scenarios in later sections.

"Wilbur's Bike Shop" sells bicycles and makes information about their company available on the Internet and on a corporate intranet. Wilbur's use XML for web publication and information exchange. The information collected by Wilbur's is spread across several web resources:

- a product catalogue – containing product name, product code, price and description;
- advertisements – containing product name, price and description;
- customer reports – listing the products purchased by particular customers;
- service reports – giving problems with products reported by customers.

Wilbur's has only one product catalogue, but many advertisements, customer reports and service reports. Figure 3.2 gives an overview of Wilbur's resources. The information is distributed across different web servers, and edited by different staff members. Examples of each type of file can be found in Appendix A.

It should be clear that much of this information, though produced independently, is closely related. For example: the product names in the advertisements and those in the catalogue;

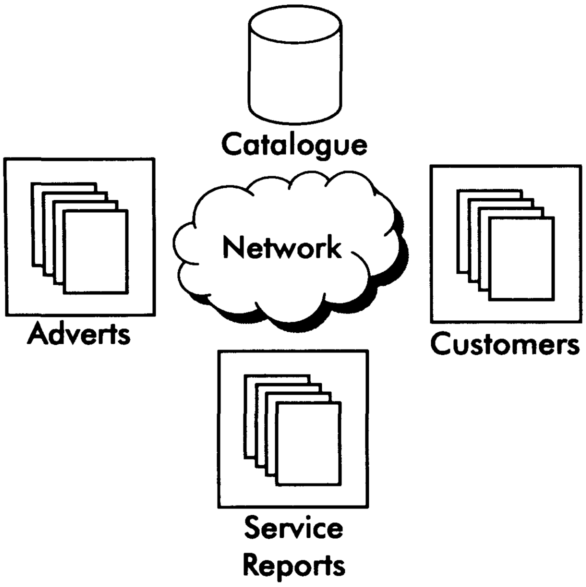


Figure 3.2: Wilbur's resources

the advertised prices and the product catalogue prices; the products listed as sold to a customer and those in the product catalogue; the goods reported as defective in the service reports and those in the customer reports; and so on.

Relationships among independently evolving and separately managed resources can give rise to inconsistencies. This is not necessarily a bad thing but it is important to be aware of such inconsistencies and deal with them appropriately. In view of this, Wilbur's would like to check their resources to establish their position. Some of the questions Wilbur's would like to answer include:

- *Are all the products that are listed in Adverts also present in the product catalogue?*
- *Do the advertised prices and the catalogue prices correspond?*
- *Are the products listed as sold to a customer in the product catalogue?*
- *Did we sell goods reported as defective to the customer reporting the problem?*

In the remainder of the thesis we will demonstrate how consistency of Wilbur's various data sources can be managed in a structured fashion; how their constraints can be expressed formally and how they can be checked; and how problems can be reported and presented to the user.

3.4 Chapter Summary

This overview has introduced the workings of our proposed consistency checking service: we translate documents into XML for the purpose of a check; we use a language that can express inter-document constraints to constrain the documents; and we evaluate constraints to provide *consistency links* that link together inconsistent elements in documents.

We have discussed the role of XML as a mechanism for overcoming syntactic heterogeneity. It allows us to dispense with syntactic integration problems and concentrate on the more difficult problem of semantic heterogeneity. At the same time, we have argued that the approach proposed in this thesis builds on very basic mechanisms such as unique identifiers and element addressing, and is thus not dependent on XML.

The final section has introduced a running example that will be used from this point forward to facilitate the discussion of novel concepts before turning to more complex examples.

With this overview in place, the following chapter takes us through some necessary groundwork that will lead to a formal definition of our document and path model, which we will later use in the definition of our novel evaluation semantics.

4 A Model for Documents and Constraints

In this chapter we provide the foundations for the formal investigation of our consistency checking semantics, by specifying a model for documents, paths in documents, and a constraint language. The chapter can serve as a rough overview of our model of documents on first reading, and as a reference for the semantic definition on more detailed reading.

Our consistency checking service falls back onto a common syntactic representation for documents during the course of the check. We have chosen XML as a convenient way of representing documents. In order to manipulate XML in memory we make use of the Document Object Model (DOM) [Apparao et al., 1998].

The DOM is a model for the in-memory representation of XML documents as trees. Using it to represent documents brings with it a number of advantages: it gives us immediate access to the already rich set of languages expressed in XML; it lets us use related standards like XPath [Clark and DeRose, 1999] for accessing elements and manipulating the documents; and it lets us use XML parsers like Xerces [Apache Software Foundation, 2003] to validate and load documents. Potential disadvantages include the relatively large size of DOM trees, due to the extra information like namespace prefixes that could be discarded from syntax trees, and perhaps the lack of type information. We decided that neither of these problems outweigh the significant advantages of choosing an open representation.

From this point onward, the thesis makes frequent reference to the DOM and XPath. In particular, we will be referring to these standards in the definition of our constraint language at the end of this chapter. We thus include a simplified abstract model of both, and a basic semantics for the evaluation of XPath.

The model and semantics borrow heavily from [Wadler, 1999], reusing most of the set names and functions for the DOM and some of the definition of XPath. It has been necessary to repeat many of the data type and function definitions in order to be able to add crucial missing features such as support for variables. The differences between our model and semantics, and Wadler's paper, as well as features from the XPath specification that we have omitted to simplify the discussion, will be highlighted in the appropriate sections.

4.1 Basic Data Types

XPath and the DOM use a number of basic data types. We have added *Number* here to what is defined in [Wadler, 1999] in order to make our coverage of XPath more complete.

Definition 4.1. Σ is an alphabet.

Definition 4.2. $String = \Sigma^*$ is the set of all strings over Σ. ε is the empty string.

Definition 4.3. $Variable \subset String$ is the set of all legal variable names. We will leave this unspecified, but assume it is a subset of the set of strings.

Definition 4.4. *Number* is the set of decimal numbers. We use this set name in order to stay in line with the XPath specification.

Definition 4.5. *Boolean* is the set $\{\top, \bot\}$ where $\top$ represents the truth value *true* and $\bot$ represents the truth value *false*.

4.2 Document Object Model

We will now define a simplified abstract model of a DOM tree that will later help us to define a semantics for XPath. It will also assist us in defining the check semantics for our constraint language.

DOM trees in our model consists of a set of nodes, represented by the data type *Node*:

Definition 4.6. *Node* is the set of all nodes in DOM trees.

Definition 4.7. $NodeSet = \wp(Node)$ is the set of all sets of nodes.

We will assume that nodes are implicitly uniquely identified by their location in memory. When we compare two nodes for identity using $=$, we are thus comparing their unique identifiers. For a pair of nodes x and y, $x = y$ if and only if x is in fact referring to the same node as y.

```
<Catalogue>
  <Product>
    <Price currency="GBP">20</Price>
  </Product>
</Catalogue>
```

Figure 4.1: Sample XML Document

A node is either an *element* node, which may contain further nodes, a *text* node or an *attribute* node. In Figure 4.1, `Catalogue` is an element node, and is also the *root node*, `currency` is an attribute node, and the content of `Price` is a text node. Every node, except the root node, has exactly one parent node. For any given node, exactly one of these functions returns $\top$:

$$\begin{aligned} isElement &: Node \rightarrow Boolean \\ isAttribute &: Node \rightarrow Boolean \\ isText &: Node \rightarrow Boolean \end{aligned}$$

isElement returns $\top$ if and only if its parameter is an element node, and $\perp$ otherwise. *isAttribute* and *isText* work the same way for attribute and text nodes, respectively.

The following function returns $\top$ only for the root node:

$$isRoot : Node \rightarrow Boolean$$

Nodes in the tree are related using the following basic functions:

$$\begin{aligned} parent &: Node \rightarrow NodeSet \\ children &: Node \rightarrow NodeSet \\ attributes &: Node \rightarrow NodeSet \\ root &: Node \rightarrow Node \end{aligned}$$

parent returns a set containing the parent node of a node, or $\emptyset$ if the node is the root node – a node can never have more than one parent node, we return a set to make the following definitions easier. *children* returns a set containing all element or text child nodes of a node or $\emptyset$ if there are none. *attributes* returns the set of all attribute nodes of an element node – in the DOM, attributes are not children of a node, but are treated separately, – or $\emptyset$ if there are none. Finally, *root* returns the root node of the document that the node is contained in. We use these basic functions to define two useful additional functions:

$$\begin{aligned} descendants &: Node \rightarrow NodeSet \\ descendants(n) &= children(n) \cup \{x \mid x = descendants(y),\ y \in children(n)\} \end{aligned}$$

$$\begin{aligned} siblings &: Node \rightarrow NodeSet \\ siblings(n) &= \begin{cases} (children(p) \setminus n \mid \{p\} = parent(n)), & parent(n) \neq \emptyset \\ \emptyset,\ otherwise \end{cases} \end{aligned}$$

descendants returns all text and element nodes reachable from the parameter node by transitively applying *children*. *siblings* returns the set containing all nodes that have the same parent as the parameter node, excluding the node itself, or $\emptyset$ if there are no such nodes.

Several invariants constrain the relationship of the functions defined above. Root nodes have no parents and therefore also have no siblings:

$$\forall n \in Node(isRoot(n) \rightarrow parent(n) = \emptyset)$$
$$\forall n \in Node(isRoot(n) \rightarrow siblings(n) = \emptyset)$$

A node is a parent of another if and only if that node is a child or attribute of the node:

$$\forall x, y \in Node(parent(x) = \{y\} \longleftrightarrow (x \in children(y) \vee x \in attributes(y)))$$

Only element nodes have attributes or children:

$$\forall n \in Node(children(n) \neq \emptyset \rightarrow isElement(n))$$
$$\forall n \in Node(attributes(n) \neq \emptyset \rightarrow isElement(n))$$

Every node in a DOM tree has a name and a value. We will access these using the two functions *name* and *value*.

Definition 4.8. *name* : *Node* $\rightarrow$ *String* returns the name of a node. The name depends on the type of the node:

- Element nodes – the name is the tag name of the element.
- Attribute nodes – the name is the attribute name.
- Text nodes – the name is ε.

Definition 4.9. *value* : *Node* $\rightarrow$ *String* returns the value of a node. The value depends on the type of the node:

- Element nodes – the value is the concatenated value of all child nodes that are text nodes. In practice, the child nodes are traversed in document order. For this discussion, the order is insignificant.
- Attribute nodes – the value is the attribute value.
- Text nodes – the value is the text content.

Looking back at Figure 4.1, the name of the root node is "Catalogue" and its value is ε. The name of the price element's attribute is "currency" and its value is "GBP". The name of the text node contained in the price element is ϵ and its value is "20". The value of the price element itself is also "20", since that is the concatenated value of all its text node children.

In summary, we have defined a DOM tree as a set of nodes that are related using a number of navigational functions. Each node in the tree has a name and a value, and is uniquely identified by its location in memory. The definition deliberately omits syntactic details such as XML namespaces, and a plethora of navigational functions, because neither are relevant to the following discussion.

4.3 XPath Abstract Syntax

XPath [Clark and DeRose, 1999] is a language for accessing the contents of XML documents. When applied to DOM trees, it can be used for specifying paths in the trees in order to identify or retrieve sets of nodes. It supports simple expressions that can be used to combine basic data types and node values to compute a result.

As an example, /Catalogue/Product points to all element nodes named Product that are children of all nodes named Catalogue, which also has to be the root node. Applying this XPath expression to our sample document in Figure 4.1 in the previous section would select a node set containing a single Product element. XPath also supports navigation other than downward navigation to child nodes. The direction of navigation is called an *axis* in XPath. For example, /Catalogue/Product/sibling::* makes use of the sibling axis to find all siblings of the Product element. In the case of our example, this would return an empty set.

The basic navigational unit of XPath is called a *step* and consists of an axis, which defaults to the child axis, and a node test. For example, sibling::* is a step, sibling is the axis and * is a wildcard node test. A step is evaluated relative to a *context*, which is the current retrieved set of nodes. Before evaluating a path, the context is empty. When evaluation is complete, the context becomes the result set. For example, before evaluating /Catalogue/Product the context is $\emptyset$. After the first step, it changes to {Catalogue}. The step Product, which selects all child nodes called Product is now applied to this context and the resulting new context, {Product} is returned as the result.

XPath also features an expression language that contains the boolean logical connectives, arithmetic operators, and an extensive string manipulation and arithmetic function library. An XPath *expression*, which may be a path, a literal or a combination of a number of paths and literals using arithmetic operators, can be evaluated to return either a string, number or boolean value. XPath makes use of these expressions in *predicates* that are used to restrict the nodes selected at each step in a path. For example, `/Catalogue/Product[@currency="GBP"]` selects only those `Product` elements whose `currency` attribute equals `GBP`. In other words, predicates are used to filter the context node set after each step.

In the remainder of the thesis we will frequently have to refer to XPath expressions as part of formal definitions. We have defined a simplified abstract syntax of XPath that will enable us to concentrate on the basic features without having to worry about syntactic sugar. The abstract syntax discards a number of features of XPath that are irrelevant to our discussion: we use the generic term `function` for XPath functions, rather than defining the whole function library, the expression language is significantly reduced and only includes the basic arithmetic and boolean operators, and we restrict the navigational axes to parent, child, ancestor, sibling and descendant.

Definition 4.10. XPath Abstract Syntax

$$
\begin{array}{lcl}
s : \mathit{String} & & \\
v : \mathit{Variable} & & \\
n : \mathit{Number} & & \\
p : \mathit{Path} & ::= & /p \mid //p \mid \$v \mid \$v/p \mid \$v//p \mid p_1|p_2 \mid p_1/p_2 \mid p_1//p_2 \mid p[e] \mid \\
 & & s \mid * \mid @s \mid @* \mid \mathit{text}() \mid \\
 & & \mathit{ancestor} :: s \mid \mathit{ancestor} :: * \mid \mathit{sibling} :: s \mid \mathit{sibling} :: * \mid \\
 & & . \mid .. \\
e : \mathit{Expr} & ::= & e_1 + e_2 \mid e_1 - e_2 \mid e_1 \ \mathit{and}\ e_2 \mid e_1 \ \mathit{or}\ e_2 \mid \\
 & & \mathit{not}\ e \mid e_1 = e_2 \mid s \mid n \mid p \mid \mathit{function}(e)
\end{array}
$$

We will now give a meaning to this abstract syntax using a denotational semantics and the definition of DOM trees given in the preceding section.

4.4 XPath Semantics

This section introduces a denotational semantics for XPath that defines how paths and expressions are evaluated over a DOM tree. We will take as our input an expression, referred to as *Expr* in the abstract syntax, a "context node" that defines where the evaluation starts, and a variable context that contains bindings of variable names to previously retrieved nodes. We will evaluate the expression to a member of the set *Result*, which is the set of possible results of evaluating an XPath expression:

Definition 4.11. $\mathit{Result} = \mathit{String} \cup \mathit{Number} \cup \mathit{Boolean} \cup \mathit{NodeSet}$.

It will be necessary to convert results of path evaluations to specific data types. For example, when evaluating a predicate on a particular step we need to convert the result of evaluating an expression to a boolean in order to determine whether to include the step's node in the result. These conversions will be performed by the functions *toString*, *toBoolean* and *toNumber*; we have no use for a function *toNodeSet*.

Definition 4.12. Conversion Functions

$$\begin{array}{lcl} \mathit{toString} & : & \mathit{Result} \rightarrow \mathit{String} \\ \mathit{toNumber} & : & \mathit{Result} \rightarrow \mathit{Number} \\ \mathit{toBoolean} & : & \mathit{Result} \rightarrow \mathit{Boolean} \end{array}$$

We will define *toString* informally as follows: strings are converted to strings using the identity function; to convert a number the function concatenates the string representation of each digit in the number and inserts a decimal point as appropriate if the fractional part is non-zero, e.g. 5 becomes `"5"` and 3.2 becomes `"3.2"`; booleans are converted as follows: $\top$ becomes `"true"` and $\perp$ becomes `"false"`; and finally, node sets are converted by concatenating the result of applying *value* (the string value, Definition 4.9) to each node in the set.

toNumber is defined as follows: strings that are representations of numbers are converted to the numbers they represents, otherwise they are converted to 0. Numbers are converted using the identity function; booleans are converted as follows: $\top$ becomes 1 and $\perp$ becomes 0; finally, for a node set n, $\mathit{toNumber}(n) = |n|$, that is we return the cardinality of the set.

Finally, we will define *toBoolean*: for any String s, $\mathit{toBoolean}(s) = (s = \text{“}\mathit{true}\text{”})$; for any number n, $\mathit{toBoolean}(n) = (n \neq 0)$, therefore $\mathit{toBoolean}(5) = \top$ but $\mathit{toBoolean}(0) = \perp$; booleans are converted into booleans using the identity function; and for any node set n, $\mathit{toBoolean}(n) = (n \neq \emptyset)$, that is the result is true if and only if the set is non-empty. This concludes the definition of results and conversion functions. We now need to introduce variables.

Paths can be relative to variables, where a variable is an identifier for a node set. For example `$x/Product` would select the union of all `Product` elements that are children of the nodes in the set of nodes identified by `x`. In order to handle variable assignment and lookup, we need to define a binding context and the two functions *bind* and *lookup*.

Definition 4.13. A *binding context* is a function $\rho : \mathit{BindingContext} = \mathit{Variable} \rightarrow \mathit{NodeSet}$ that maps a variable name to a set of nodes. For a given variable name v, $\rho(v)$ returns the set of nodes to which v has been bound or $\emptyset$ if it has not been bound. Since ρ is a function, duplicate bindings for the same variable name are not possible.

Definition 4.14. *bind* is a function that introduces a new variable binding into a binding context. It is a higher-order function that takes an existing binding context as a parameter:

$$\begin{array}{lcl} \mathit{bind} & : & \mathit{Variable} \times \mathit{NodeSet} \times \mathit{BindingContext} \rightarrow \mathit{BindingContext} \\ \mathit{bind}(v, n, \rho) & = & \rho_1, \mathit{where}\ \rho_1(v_1) = \mathit{if}\ (v_1 = v)\ \mathit{then}\ n\ \mathit{else}\ \rho(v_1) \end{array}$$

Definition 4.15. *lookup* is used to look up variables in a binding context. We introduce this simply to make the semantics more readable:

$$\begin{array}{lcl} \mathit{lookup} & : & \mathit{Variable} \times \mathit{BindingContext} \rightarrow \mathit{NodeSet} \\ \mathit{lookup}(v, \rho) & = & \rho(v) \end{array}$$

We can now define the main evaluation function $\mathcal{S}$, which takes a path, a context node and a binding context and returns a node set that contains the result of evaluating the path. We will also define the function $\mathcal{E}$, which evaluates an expression given a context node and binding context and returns a *Result.* The signature of these functions is:

$$\begin{array}{lcl} \mathcal{S} & : & \mathit{Path} \rightarrow \mathit{Node} \rightarrow \mathit{BindingContext} \rightarrow \mathit{NodeSet} \\ \mathcal{E} & : & \mathit{Expr} \rightarrow \mathit{Node} \rightarrow \mathit{BindingContext} \rightarrow \mathit{Result} \end{array}$$

When defining the functions we will use the semantic notation $\mathcal{S}[\![p]\!]_{n,\rho}$ where p is the syntax constract being defined, and the subscript parameters represent the environment necessary for the definition.

The following examples illustrate the definition that follows below. $\mathcal{S}[\![//p]\!]_{n,\rho}$ selects all nodes that match the path p anywhere in the tree by making all descendants of the root element the context for evaluating p. The path $p_1 \mid p_2$ is evaluated by evaluating p_1 and p_2 separately and then taking the union of the resulting node sets. p_1/p_2 is a straightforward child axis step and is evaluated by evaluating p_1 over the context node, and then making each node in the result of this evaluation the new context node and evaluating p_2. $\mathcal{S}[\![p[e]]\!]_{n,\rho}$ selects a set of nodes using the path p and then uses the function $\mathcal{E}[\![e]\!]$, which we will define next, to filter the resulting node set – only nodes for which the expression e is true remain in the set. $\mathcal{S}[\![s]\!]_{n,\rho}$ is a terminal point for the recursion of $\mathcal{S}$ and selects all children of the context nodes whose name matches the string s. $\mathcal{S}[\![@*]\!]_{s,p}$ selects all attribute nodes of the context node. The next few definitions provide navigations over the sibling and ancestor axes, and finally $\mathcal{S}[\![.]\!]_{n,p}$ selects the context node and $\mathcal{S}[\![..]\!]_{n,p}$ selects the parent node of the context node.

Definition 4.16. Path Selection

$$
\begin{array}{lcl}
\mathcal{S} & : & \mathit{Path} \rightarrow \mathit{Node} \rightarrow \mathit{BindingContext} \rightarrow \mathit{NodeSet} \\
\mathcal{S}\llbracket /p \rrbracket_{n,\rho} & = & \mathcal{S}\llbracket p \rrbracket_{root(n),\rho} \\
\mathcal{S}\llbracket //p \rrbracket_{n,\rho} & = & \{n_2 \mid n_1 \in \mathit{descendants}(root(n)),\ n_2 \in \mathcal{S}\llbracket p \rrbracket_{n_1,\rho}\} \\
\mathcal{S}\llbracket \$v \rrbracket_{n,\rho} & = & \mathit{lookup}(v,\rho) \\
\mathcal{S}\llbracket \$v/p \rrbracket_{n,\rho} & = & \{n_2 \mid n_1 \in \mathit{lookup}(v,\rho),\ n_2 \in \mathcal{S}\llbracket p \rrbracket_{n_1,\rho}\} \\
\mathcal{S}\llbracket \$v//p \rrbracket_{n,\rho} & = & \{n_3 \mid n_1 \in \mathit{lookup}(v,\rho),\ n_2 \in \mathit{descendants}(n1),\ n_3 \in \mathcal{S}\llbracket p \rrbracket_{n_2,\rho}\} \\
\mathcal{S}\llbracket p_1 | p_2 \rrbracket_{n,\rho} & = & \mathcal{S}\llbracket p_1 \rrbracket_{n,\rho} \cup \mathcal{S}\llbracket p_2 \rrbracket_{n,\rho} \\
\mathcal{S}\llbracket p_1/p_2 \rrbracket_{n,\rho} & = & \{n_2 \mid n_1 \in \mathcal{S}\llbracket p_1 \rrbracket_{n,\rho},\ n_2 \in \mathcal{S}\llbracket p_2 \rrbracket_{n_1,\rho}\} \\
\mathcal{S}\llbracket p_1//p_2 \rrbracket_{n,\rho} & = & \{n_3 \mid n_1 \in \mathcal{S}\llbracket p_1 \rrbracket_{n,\rho},\ n_2 \in \mathit{descendants}(n_1),\ n_3 \in \mathcal{S}\llbracket p_2 \rrbracket_{n_2,\rho}\} \\
\mathcal{S}\llbracket p[e] \rrbracket_{n,\rho} & = & \{n_1 \mid n_1 \in \mathcal{S}\llbracket p \rrbracket_{n,\rho},\ \mathit{toBoolean}(\mathcal{E}\llbracket e \rrbracket_{n_1,\rho}) = \top\} \\
\mathcal{S}\llbracket s \rrbracket_{n,\rho} & = & \{n_1 \mid n_1 \in \mathit{children}(n),\ \mathit{name}(n_1) = s\} \\
\mathcal{S}\llbracket * \rrbracket_{n,\rho} & = & \{n_1 \mid n_1 \in \mathit{children}(n),\ \mathit{isElement}(n)\} \\
\mathcal{S}\llbracket @s \rrbracket_{n,\rho} & = & \{n_1 \mid n_1 \in \mathit{attributes}(n),\ \mathit{name}(n_1) = s\} \\
\mathcal{S}\llbracket @* \rrbracket_{n,\rho} & = & \{n_1 \mid n_1 \in \mathit{attributes}(n)\} \\
\mathcal{S}\llbracket \mathit{text}() \rrbracket_{n,\rho} & = & \{n_1 \mid n_1 \in \mathit{children}(n),\ \mathit{isText}(n_1)\} \\
\mathcal{S}\llbracket \mathit{ancestor} :: s \rrbracket_{n,\rho} & = & \{n_1 \mid n \in \mathit{descendants}(n_1),\ \mathit{name}(n_1) = s\} \\
\mathcal{S}\llbracket \mathit{ancestor} :: * \rrbracket_{n,\rho} & = & \{n_1 \mid n \in \mathit{descendants}(n_1)\} \\
\mathcal{S}\llbracket \mathit{sibling} :: s \rrbracket_{n,\rho} & = & \{n_1 \mid n_1 \in \mathit{siblings}(n),\ \mathit{name}(n_1) = s\} \\
\mathcal{S}\llbracket \mathit{sibling} :: * \rrbracket_{n,\rho} & = & \{n_1 \mid n_1 \in \mathit{siblings}(n)\} \\
\mathcal{S}\llbracket . \rrbracket_{n,\rho} & = & \{n\} \\
\mathcal{S}\llbracket .. \rrbracket_{n,\rho} & = & \mathit{parent}(n)
\end{array}
$$

Next we will define the semantics for $\mathcal{E}$, which evaluates expressions and returns a *Result*. This function is used by $\mathcal{S}$ to evaluate predicates and we will also make use of it directly in later sections.

$\mathcal{E}$ defines an interpretation for a number of different constructs: the arithmetic operators $+$ and $-$ are handled by evaluating the parameters, turning them into numbers, and then applying the usual arithmetic functions; the boolean operators take on the usual boolean logic interpretations; equality comparison is handled by converting both parameters into strings and then comparing the strings for equality; and expressions that are simple paths are evaluated by using $\mathcal{S}$ to select a node set.

Function invocation using `function(e)` is not refined any further at this point. We expect that *function* is a function with the signature *function* : *Expr* → *Result*, but for reasons of readability, and because it is not necessary for our discussion, we do not fully introduce the concept of a function context here. We will simply assume that all functions are side effect free – that is, they do not change the document – and terminate. Many of the functions in XPath take more than one parameter, but they all return a *Result*. The XPath function library includes:

- String manipulation, e.g. *substring* : (*Expr*, *Number*) → *String*, which returns the substring of a string (or another result that has been converted to a string) starting from the given index.
- Boolean functions, e.g. *true*() : *Boolean*, which always returns ⊤.
- Node set functions, e.g. *count* : *NodeSet* → *Number* which returns the cardinality of a node set.
- Number functions, e.g. *round* : *Expr* → *Number*, which rounds the result of converting the evaluated expression to a number up to the nearest integer.

Definition 4.17. Expression Evaluation

$$\begin{array}{lcl}
\mathcal{E} & : & \mathit{Expr} \rightarrow \mathit{Node} \rightarrow \mathit{BindingContext} \rightarrow \mathit{Result} \\
\mathcal{E}[\![e_1 + e_2]\!]_{n,\rho} & = & \mathit{toNumber}(\mathcal{E}[\![e_1]\!]_{n,\rho}) + \mathit{toNumber}(\mathcal{E}[\![e_2]\!]_{n,\rho}) \\
\mathcal{E}[\![e_1 - e_2]\!]_{n,\rho} & = & \mathit{toNumber}(\mathcal{E}[\![e_1]\!]_{n,\rho}) - \mathit{toNumber}(\mathcal{E}[\![e_2]\!]_{n,\rho}) \\
\mathcal{E}[\![e_1 \mathit{ and } e_2]\!]_{n,\rho} & = & \mathit{toBoolean}(\mathcal{E}[\![e_1]\!]_{n,\rho}) \wedge \mathit{toBoolean}(\mathcal{E}[\![e_2]\!]_{n,\rho}) \\
\mathcal{E}[\![e_1 \mathit{ or } e_2]\!]_{n,\rho} & = & \mathit{toBoolean}(\mathcal{E}[\![e_1]\!]_{n,\rho}) \vee \mathit{toBoolean}(\mathcal{E}[\![e_2]\!]_{n,\rho}) \\
\mathcal{E}[\![\mathit{not}\ e]\!]_{n,\rho} & = & \neg \mathit{toBoolean}(\mathcal{E}[\![e]\!]_{n,\rho}) \\
\mathcal{E}[\![e_1 = e_2]\!]_{n,\rho} & = & \mathit{toString}(\mathcal{E}[\![e_1]\!]_{n,\rho}) = \mathit{toString}(\mathcal{E}[\![e_2]\!]_{n,\rho}) \\
\mathcal{E}[\![s]\!]_{n,\rho} & = & s \\
\mathcal{E}[\![n]\!]_{n,\rho} & = & n \\
\mathcal{E}[\![p]\!]_{n,\rho} & = & \mathcal{S}[\![p]\!]_{n,\rho} \\
\mathcal{E}[\![\mathit{function}(e)]\!]_{n,\rho} & = & \mathit{function}(e)
\end{array}$$

This concludes our definition of XPath and its associated evaluation semantics. We have defined the two types *Path*, which represents a path in a document, and *Expr*, which combines paths, literals, numbers, boolean operators, arithmetic operators and functions into a simple expression language. The two semantic functions $\mathcal{S}$, which selects a set of nodes given a *Path*, and $\mathcal{E}$, which evaluates an expression, define the meaning for these constructs.

In the following section we will extend the definitions of $\mathcal{S}$ and $\mathcal{E}$ to handle path evaluation over a forest of DOM trees. We will then use these extended definitions in our constraint language semantics.

DOM and XPath Summary

- *String*, *Number* and *Boolean* are the basic data types.
- A DOM tree consists of a set of nodes, represented by *Node*. A node is either an element node, a text node or an attribute node.
- Every node has a unique identity, a name and a value.
- XPath provides paths for selecting nodes from documents and expressions for simple computations and logical comparisons.
- Paths are either absolute, or relative to variables and are evaluated to a node set using the semantic function $\mathcal{S}$.
- Expressions evaluate to a string, number, boolean or node set using the semantic function $\mathcal{E}$. Conversion functions can be used to convert each of these types to a string, number or boolean.
- Variables are handled using a *binding context* and the two functions *bind* and *lookup*.

4.5 Constraint Language

We have put in place the foundations to represent documents in a common format as DOM trees, and to query them for sets of nodes using XPath. We can now specify what it means for a set of documents to be consistent by relating nodes in the documents through constraints. By definition, elements that do not obey these constraints are causing an inconsistency.

The choice of constraint language is not entirely straightforward, as evaluating formulae in a language and providing suitable feedback becomes progressively more difficult as the expressive power of the language increases. An earlier approach [Zisman et al., 2000] was based on a simple language that expressed a relationship between a set of "source" and a set of "destination" elements. We have also experimented with a restricted language that expressed constraints between two sets of elements, for example "for every element in set A, there must be an element in set B". While it was easy to specify diagnostics for these languages – inconsistent elements in the sets could easily be pointed out – it soon became clear that the expressive power of such languages was insufficient for a number of application domains, most notably in software engineering. The UML [OMG, 2000a] is a good counter-example as its validation requires expressiveness beyond propositional logic. We thus chose instead a language based on first order logic as our constraint language as we were hopeful that it would be expressive enough for our purposes, a hypothesis we have since confirmed in a number of case studies.

We will introduce our constraint language using the running example. The question *"Are all the products that are listed in adverts also present in the catalogue?"* can be represented more formally as an assertion: *For all Advert elements, there exists a Product element in the Catalogue element where the ProductName subelement of the former equals the Name subelement of the latter.* If this condition holds, a consistent relationship exists between the `Advert` element and the `Product` element. Otherwise, the `Advert` element is inconsistent with respect to our rule as there is no matching `Product` element.

Definition 4.18. Constraint Language Abstract Syntax

$$\begin{array}{lcll}
p & : & \mathit{Path} & \\
e & : & \mathit{Expr} & \\
v & : & \mathit{Variable} & \\
f & : & \mathit{formula} & ::= \quad \forall v \in p\ (f) \mid \exists v \in p\ (f) \mid \\
 & & & \quad f_1 \textit{ and } f_2 \mid f_1 \textit{ or } f_2 \mid f_1 \textit{ implies } f_2 \mid \textit{not} f \mid \\
 & & & \quad e_1 = e_2 \mid \textit{same } v_1\ v_2
\end{array}$$

Definition 4.18 gives an abstract syntax of our constraint language. The language is a function-free first order predicate logic that expresses constraints over finite sets. These sets contain elements and attributes of XML documents. Since XPath permits variable references, formulae can make reference to variables bound in superformulae. This allows predicates to be used for testing properties of nodes currently bound by a quantifier. We can express our semi-formal assertion more formally in this language:

$$\forall a \in \text{/Advert } (\exists p \in \text{/Catalogue/Product (\$a/ProductName=\$p/Name)})$$

Note that the paths in the constraint refer to elements from different documents, while not explicitly referring to which documents the constraint is applied to. This is a matter for the evaluation semantics that we apply to the language. This evaluation semantics, which we will discuss in the next chapter, will extend our notion of path evaluation to multiple documents.

To ensure constraint integrity, we place some additional static semantic constraints on formulae in the language that cannot be captured in the abstract syntax. We will state these informally here:

Definition 4.19. *For any quantifier formula, the variable v must not be equal to a variable bound in an outer formula.* This is to prevent double bindings, which would lead to accidental overwriting of the binding context.

Definition 4.20. *If a path expression is relative to a variable v, that variable must have been bound by a quantifier in an outer formula.*

Definition 4.21. *For formulae of the form $e_1 = e_2$, any path expressions contained in e_1 or e_2 must be relative to a variable.* For example, $a/ProductName=$p/Name is legal, while /Advert/ProductName=$p/Name is not. We made this choice because we wanted to restrict distributed path evaluation over multiple documents to quantifiers, to simplify implementation. It also eliminates any implicit quantification if the path points to entries in multiple documents, which would be better captured as an explicit quantifier.

Since the only iteration constructs in the language are quantifiers that iterate over finite sets of elements retrieved from documents, we cannot directly express constraints that require any form of infinity. For example, the constraint *for all elements x, the children of x are prime numbers* would require quantification over the integers to express the latter half of the constraint and thus cannot be expressed directly in this language. Such a case would have to be handled by implementing a predicate that performs a primality test, using a Turing-complete programming language. Nevertheless, as will be shown in the following, the power of the constraint language is great enough to express a wide range of static semantic constraints, including those of the Unified Modeling Language [OMG, 2000a].

Constraint Language Summary

- **The constraint language combines first order logic with XPath in order to relate nodes in DOM trees.**
- **The language includes universal and existential quantifiers, boolean connectives and predicates.**

4.6 Chapter Summary

This chapter contains the groundwork for the formal definition of our consistency checking semantics. It defines a model for XPath and the Document Object Model, and gives a semantics for evaluating paths over DOM trees.

Though most of it is based on [Wadler, 1999], we believe that this extended treatment of the DOM and XPath will be valuable as reference material in its own right for the semantic definition of other types of systems that deal with XML input.

We have also introduced the abstract syntax of a constraint language that relates elements in DOM trees. In the following chapter we will look at consistency checking as the process of evaluating constraints written in this language and providing diagnostic results. We will do this by combining the abstract syntax with the document and path models presented here.

5 Checking Semantics

In this chapter we get to the core theoretical contribution of our investigation: how to evaluate constraints between documents and provide a rich diagnostic result that enables users to identify inconsistency, and trace it effectively should it involve multiple documents.

We will first define a standard *boolean* semantics for evaluating constraints over multiple DOM models. This boolean semantics will follow the standard definition of first order logic, and adapt it to apply to a set of DOM trees.

Following an illustration of the shortcomings of the boolean semantics as a diagnostic mechanism, we will move on to defining a novel evaluation semantics for our constraint language that connects inconsistent elements in documents through hyperlinks called *consistency links*. By using a denotational semantics as a formal basis, and notions from game semantics [Hintikka and Sandu, 1996] for explanation, we cover the entire language and provide examples for each construct.

Towards the end of the chapter we will cover some of the assumptions and design decisions that went into defining our evaluation semantics. We will also investigate its relationship to the standard boolean interpretation, providing proofs that the de Morgan's laws as well as other standard equivalences hold.

5.1 Boolean Semantics

In order to clarify the meaning of the constructs of the constraint language, and to assist the later definition of our hyperlink semantics, we first define a boolean interpretation that returns *true* when a formula holds for a set of documents and *false* otherwise.

The abstract syntax in Definition 4.18 on page 45 does not contain any notion of a document, or the location of a document – it refers to elements using paths without specifying where those paths are applied. This is intentional: it makes the language location transparent and hence formulae in the language can be applied without modification when documents are relocated. Rather than hardcoding names or locations, documents are included depending on their content. This is done using the paths in the constraints: absolute paths like `/Catalogue/Product` are executed on each document, and the union of the resulting node sets is taken as the result. Paths that are relative to variables executed only on those documents that contain the nodes pointed to by the variable.

To define this more precisely, we introduce the notion of a *document set*, which represents a forest of DOM trees. A document set is a node set that contains only root nodes:

Definition 5.1. *DocumentSet* $\subset$ *NodeSet*, $\forall s \in$ *DocumentSet* $(\forall n \in s(isRoot(n)))$

We also define the functions $\mathcal{S}_d$ and $\mathcal{E}_d$, which perform XPath selection and expression evaluation by applying the previously defined functions $\mathcal{S}$ and $\mathcal{E}$ over a document set. $\mathcal{S}_d$ returns the union of applying $\mathcal{S}$ over all trees in the set. In the case of $\mathcal{E}_d$, the definition is exactly the same as for $\mathcal{E}$ for basic expression computation. When paths have to be evaluated within $\mathcal{E}_d$, $\mathcal{S}_d$ is used instead of $\mathcal{S}$. We use n_0 as a dummy context node in $\mathcal{E}_d$ where expression evaluation is required to simplify the notation. We know from Definition 4.21 (page 46) that paths in expressions will be relative to a variable anyway, and the variable will point to the context node set.

Since there will only be one document set to be checked at any given time, we will omit passing this set as a parameter to $\mathcal{S}_d$ and $\mathcal{E}_d$[1] in order to improve readability, and instead assume that the set is identified by the global variable $\mathbb{D}$.

Definition 5.2. $\mathbb{D}$ refers to the document set that is being checked, $\mathbb{D} \in$ *DocumentSet*

Definition 5.3. $\mathcal{S}_d$ and $\mathcal{E}_d$

$$\begin{array}{lcl} p & : & Path \\ e & : & Expr \end{array}$$

$$\begin{array}{lcl}
\mathcal{S}_d & : & Path \rightarrow BindingContext \rightarrow NodeSet \\
\mathcal{S}_d[\![p]\!]_\rho & = & \bigcup_{d_i \in \mathbb{D}}(\mathcal{S}[\![p]\!]_{d_i,\rho}) \\
 & & \\
\mathcal{E} & : & Expr \rightarrow BindingContext \rightarrow Result \\
\mathcal{E}_d[\![e]\!]_\rho & = & \mathcal{S}_d[\![e]\!]_\rho,\ if\ e \in Path \\
 & = & \mathcal{E}[\![e]\!]_{n_0,\rho},\ otherwise
\end{array}$$

As an example, suppose our document set is the set of documents owned by Wilbur's bike shop that we have included in Appendix A. Assuming that ρ_0 is the empty binding context, $\mathcal{S}_d[\![/Advert]\!]_{\rho_0}$ would return the set $\{Advert\}$ where *Advert* is the root node of the advert file. $\mathcal{S}_d[\![/Advert \mid /Catalogue]\!]_{\rho_0}$ would return $\{Advert, Catalogue\}$ where *Catalogue* is the root node of the catalogue file. If there was more than one advert file then the root nodes of all advert files would be included in both node sets.

We can now proceed to specify the boolean interpretation of our constraint language: the function $\mathcal{B}$: *formula* $\rightarrow$ *BindingContext* $\rightarrow$ *Boolean* returns true if and only if a formula holds over all documents in the document set. $\mathcal{B}$ is normally invoked with a formula and the empty binding context ρ_0 as a parameter. ρ_0 maps all possible variable names to the empty set.

[1] Note that the subscript d in $\mathcal{S}_d$ and $\mathcal{E}_d$ is a label to differentiate the functions from $\mathcal{S}$ and $\mathcal{D}$ rather than a parameter.

```
<document>
  <elementA>15</elementA>
  <elementB>15</elementB>
  <elementC>12</elementC>
</document>
```

Figure 5.1: Example Document for $=$ and *same*

Definition 5.4 below defines the boolean semantics. The definition follows the usual interpretation of the quantifiers $\forall$ and $\exists$ and the boolean connectives. We will briefly examine the way in which each formula construct is evaluated: $\forall$ is evaluated by first evaluating the path expression, then binding the variable to each node in the resulting set in turn, using the subformula to evaluate the result given the new binding context. The overall result is *true* if and only if the subformula returns *true* for all assignments to the variable. $\exists$ is evaluated similarly, but the result is true if and only if there is at least one variable assignment for which the subformula is true. *and*, *or*, *implies* and *not* use the normal boolean connectives to combine the result of evaluating the subformulas.

$=$ and *same* are the two predicates in the language, and their interpretation differs. $=$ is intended as a value comparison whereas *same* compares the identity of nodes in the tree. Consequently, $=$ uses $\mathcal{E}$ to evaluate the two expressions, converts the resulting value into strings and compares those strings. By contrast, *same* uses the binding context to look up two node sets and checks whether these node sets contain exactly the same nodes.

To illustrate the difference between $=$ and *same*, consider the following example: assume the variable `x` is bound to `elementA` in the document shown in Figure 5.1 and `y` is bound to `elementB`. Then $\$x = \$y$ is evaluated as follows: $\mathcal{E}_d[\![\$x]\!]_\rho$ is evaluated by $\mathcal{S}_d[\![\$x]\!]_\rho$, since $\$x$ is a path expression. $\mathcal{S}_d[\![\$x]\!]_\rho$ uses *lookup* to find the current assignment to x and returns the node set $\{elementA\}$. *toString* is then applied to this node set, which takes the concatenation of *value* applied to each node. The value of *elementA* is 15, since that is the concatenation of the values of all text node children. Therefore $\mathcal{E}_d[\![\$x]\!]_\rho = 15$, and similarly $\mathcal{E}_d[\![\$y]\!]_\rho = 15$. It follows that $\$x = \$y$ evaluates to $\top$. The evaluation of *same*, however, proceeds by looking up the node sets bound to $\$x$ and $\$y$ and compares the nodes directly. Since $lookup(x, \rho) = \{elementA\}$ and $lookup(y, \rho) = \{elementB\}$, and clearly $\{elementA\} \neq \{elementB\}$, *same* returns $\bot$ for these two parameters. The following definition of $\mathcal{B}$ defines the full boolean interpretation for $formula$.

Definition 5.4. Constraint Language Boolean Interpretation

$$\begin{array}{lcl}
p & : & \mathit{Path} \\
e & : & \mathit{Expr} \\
v & : & \mathit{Variable} \\
f & : & \mathit{Formula}
\end{array}$$

$$\begin{array}{lcl}
\mathcal{B} & : & \mathit{formula} \rightarrow \mathit{BindingContext} \rightarrow \mathit{Boolean} \\
\mathcal{B}[\![\forall v \in p\ (f)]\!]_\rho & = & \bigwedge_{n_i \in \mathcal{S}_d[\![p]\!]_\rho} \mathcal{B}[\![f]\!]_{bind(v,\{n_i\},\rho)} \\
\mathcal{B}[\![\exists v \in p\ (f)]\!]_\rho & = & \bigvee_{n_i \in \mathcal{S}_d[\![p]\!]_\rho} \mathcal{B}[\![f]\!]_{bind(v,\{n_i\},\rho)} \\
\mathcal{B}[\![f_1\ \mathit{and}\ f_2]\!]_\rho & = & \mathcal{B}[\![f_1]\!]_\rho \wedge \mathcal{B}[\![f_2]\!]_\rho \\
\mathcal{B}[\![f_1\ \mathit{or}\ f_2]\!]_\rho & = & \mathcal{B}[\![f_1]\!]_\rho \vee \mathcal{B}[\![f_2]\!]_\rho \\
\mathcal{B}[\![f_1\ \mathit{implies}\ f_2]\!]_\rho & = & \mathcal{B}[\![f_1]\!]_\rho \rightarrow \mathcal{B}[\![f_2]\!]_\rho \\
\mathcal{B}[\![\mathit{not}\ f]\!]_\rho & = & \neg\mathcal{B}[\![f]\!]_\rho \\
\mathcal{B}[\![e_1 = e_2]\!]_\rho & = & \mathit{toString}(\mathcal{E}_d[\![e_1]\!]_\rho) = \mathit{toString}(\mathcal{E}_d[\![e_2]\!]_\rho) \\
\mathcal{B}[\![\mathit{same}\ v_1\ v_2]\!]_\rho & = & \mathit{lookup}(v_1,\rho) = \mathit{lookup}(v_2,\rho)
\end{array}$$

We will take a look at one more example, which applies $\mathcal{B}$ to a formula that contains a quantifier and a subformula. The formula is $\forall x \in$/document/* ($\$x = 15$). Evaluation proceeds as follows (the detailed evaluation of the equality comparisons has been omitted):

$$\begin{array}{lcl}
\mathcal{B}[\![\forall x \in /\text{document}/*\ (\$x = 15)]\!]_\rho & = & \bigwedge_{n_i}(\mathcal{B}[\![\$x = 15]\!]_{bind(x,\{n_i\},\rho)}) \mid n_i \in \mathcal{S}_d[\![/\text{document}/*]\!]_{\rho_0} \\
 & = & \mathcal{B}[\![\$x = 15]\!]_{bind(x,\{elementA\},\rho_0)} \wedge \\
 & & \mathcal{B}[\![\$x = 15]\!]_{bind(x,\{elementB\},\rho_0)} \wedge \\
 & & \mathcal{B}[\![\$x = 15]\!]_{bind(x,\{elementC\},\rho_0)} \\
 & = & \top \wedge \top \wedge \bot \\
 & = & \bot
\end{array}$$

Since the value of *elementC* is not equal to 15, the formula fails. In addition to demonstrating the boolean evaluation of a formula, this example clearly demonstrates the diagnostic weakness of boolean interpretations: very little information is available at the end of the evaluation about the elements that have contributed to an inconsistency. Clearly, in this example the value of *elementC* is a cause of inconsistency, and we would like it to be identified as such. With more complex formulae that relate multiple elements, it may be a combination of elements that causes an inconsistency and in such a case the boolean evaluation may hide even more information.

If a tolerant approach to inconsistency is chosen, it may even be irrelevant if the formula as a whole evaluates to false: take as an example a constraint that checks a property of several hundred entries in a product catalogue. If one or two of the products are inconsistent, this may well be tolerable. The overall failure of the constraint is irrelevant, what is important is that we can identify and monitor those items that are inconsistent so we can defer resolution to a later stage, or decide not to act at all.

It is these shortcomings of the boolean interpretation that we will address in the next section on link generation.

Boolean Semantics Summary

- **During formula evaluation, paths are executed over a set of documents. The result is the union of the node sets selected from the documents.**
- **The functions $\mathcal{S}_d$ and $\mathcal{E}_d$ extend $\mathcal{S}$ and $\mathcal{E}$ from the previous chapter to distributed documents.**
- **The semantic function $\mathcal{B}$ returns $\top$ if a formula holds for a document set $\mathbb{D}$, or $\bot$ otherwise.**

5.2 Link Generation Semantics

The example at the end of the last section demonstrates that the expressive power of a constraint language is only one factor in determining its usefulness in practice; it is equally important to be able to produce good diagnostics for detected inconsistencies. To draw an analogy, a good compiler does not simply state that a source file violates the static semantics of a language, but points out the line number and perhaps even identifies the statement that causes the problem. It will also specify which semantic rule has been violated. Because we have chosen to take a tolerant view of inconsistency, inconsistency does not have to be resolved immediately. It thus becomes even more important to provide precise information to document owners to enable them to monitor and pinpoint inconsistent information with a minimum of effort.

We use hyperlinks called *consistency links* to relate consistent or inconsistent elements. If a number of elements form a relationship under which a constraint is satisfied, they are connected via a *consistent link.* If they form an undesirable relationship with respect to some constraint, they are connected via an *inconsistent link.* This form of diagnosis is motivated by, and fits in well with, the use of XML as an intermediate encoding and enables us to use XLink to point exactly to the location in a document that is involved in an inconsistent relationship. It is also very suitable in the distributed setting – hyperlinks are an established mechanism for connecting distributed documents. XLink's linkbase feature allows us to retain the precise diagnostic information without making invasive document changes, since links can be maintained outside the documents.

The idea of connecting inconsistent elements using consistency links is based on earlier research [Ellmer et al., 1999]. Compared to this earlier work, which was based on a relatively restricted constraint language, the problem of finding a semantics to map from first order logic to consistency links is more complex. The remainder of this section discusses this semantics, beginning with a definition of consistency links.

5.2.1 Consistency Links

A consistency link consists of a list of nodes that we call its *locators* and a *status.* We represent locators as a list rather than a set to make it possible to connect a node to itself. This requires the node to be present multiple times in the list.

The status of a link is either *consistent*, meaning the nodes pointed to by the locators are in a consistent relationship with respect to a formula, or *inconsistent* if they are in an inconsistent relationship. What exactly it means for a node to be included in the list of locators will become clear from the semantics below.

Every locator points to exactly one node in a DOM tree. For notational convenience, we will not distinguish here between nodes and *references* to nodes – we have previously defined nodes only within the context of a DOM tree, but this slight informality does not affect the discussion. Links, and hence consistency relationships, are not restricted to connecting two elements, but can form relationships between n elements, where $n \geq 0$. Using the notation $[A]$ in type signatures to denote lists of elements of type A, $[]$ as the empty list, and $[a, b, c]$ as a list containing the nodes a, b and c, we define consistency links as follows:

Definition 5.5. $Locators = [Node]$

Definition 5.6. $Status = \{Consistent, Inconsistent\}$

Definition 5.7. $Link = Status \times Locators$

We have thus defined a consistency link as a tuple consisting of a status and a list of locators. Here are some sample links over the elements of Figure 5.1 on page 49: Assume that a rule states that all elements have to have the value 15. Then we can use $(Inconsistent, [elementC])$ as a link that identifies *elementC* as an inconsistent node. Assume another rule states that all pairs of elements in the document must have distinct values. Then $(Inconsistent,\ [elementA, elementB])$ might express that *elementA* is inconsistent with *elementB*. Finally, we could have a constraint that at least one element must have the value 361. Then a link of the form $(Inconsistent, [])$, with no locators, would represent that we could not find such an element.

One of the consequences of holding locators in lists, which are ordered constructs, rather than sets is that the locators $[elementA, elementB]$ differ from $[elementB, elementA]$. It will be useful to distinguish between these cases, because as we will see, the order of the locators will be tied to the formula constructs that create them. If an application, for example our reporting tool, wishes to treat these links as equivalent, it can still strip one of them out.

Our evaluation semantics will make use of several auxiliary functions to manipulate consistency links. The first of these functions is *flip*, which inverts the status of the links in a set of links. This should be self-explanatory from the definition:

Definition 5.8. $flip : \wp(Link) \rightarrow \wp(Link)$

$$\begin{array}{rcl} flip(links) & = & \{fliplink(l)\},\ where\ l \in links\ and \\ & & \quad fliplink((Consistent, locs)) = (Inconsistent, locs) \\ & & \quad fliplink((Inconsistent, locs)) = (Consistent, locs) \end{array}$$

Next, the function *join* combines a link l with a set of links: the locators of l are concatenated with the locators of each link in the set in turn, forming a new set of links. Furthermore, each link in the new set assumes the consistency status of l. If the set of links passed as a parameter is empty, a set containing only l parameter is returned. For any two lists a and b, let $a+b$ be the concatenated list containing all elements of a followed by all elements of b. Then:

Definition 5.9. $join : Link \times \wp(Link) \rightarrow \wp(Link)$

$$join((s_1, l_1), links) = \begin{cases} \{(s_1, l_1 + l_2) \mid (s_2, l_2) \in links\}, & links \neq \emptyset \\ \{(s_1, l_1)\}, & otherwise \end{cases}$$

Example 1. Here are some examples of how *join* works: assume L is the set of links

$$\{(Consistent, [elementA]), (Inconsistent, [elementA, elementB])\}$$

Then $join((Consistent, [elementC]), L)$ returns

$$\{(Consistent, [elementC, elementA]), (Consistent, [elementC, elementA, elementB])\}$$

For an empty set of links:

$$join((Consistent, [elementC]), \emptyset) = \{(Consistent, [elementC])\}$$

Our semantics will make heavy use of *join*. In order to prevent cluttering we will use the symbol $\circ$ as its infix representation:

Definition 5.10. $link \circ links = join(link, links)$

Finally, we extend the notion of *join* to sets. The function *setjoin* combines two sets of links by applying join to each link in the first set and the entire second set. If either set is empty, the other set is returned as the result. If both sets are empty, the result is empty:

Definition 5.11. *setjoin* : $\wp(Link) \times \wp(Link) \rightarrow \wp(Link)$

$$setjoin(l_1, l_2) \quad = \quad \begin{cases} \bigcup_{l_i \in l_1}(l_i \circ l_2), & l_1 \neq \emptyset \wedge l_2 \neq \emptyset \\ l_1, & l_1 \neq \emptyset \wedge l_2 = \emptyset \\ l_2, & l_1 = \emptyset \wedge l_2 \neq \emptyset \\ \emptyset, & otherwise \end{cases}$$

Similar to *join*, we define the infix operator $\circ_s$ as a more convenient representation of *setjoin*:

Definition 5.12. $l_1 \circ_s l_2 = setjoin(l_1, l_2)$

Example 2. If $l_1 = \{(Consistent, [elementA]), (Inconsistent, [])\}$ and $l_2 = \{Consistent, [elementB]\}$: $l_1 \circ_s l_2 = \{(Consistent, [elementA, elementB]), (Inconsistent, [elementB])\}$

Consistency Link Summary

- A *consistency link* consists of a status and a list of *locators*. The status is either *Consistent* or *Inconsistent*. The locators point to nodes in DOM trees.
- *flip* inverts the status of all links in a set of links.
- *join* prepends the locators of a link to all links in a set of links and overrides their status with that of the single link. $\circ$ is the infix form of *join*.
- *setjoin* takes two sets of links and applies join to each link in the first set and the entire second set. $\circ_s$ is the infix form of *setjoin*.

5.2.2 Link Generation Semantics

The following definition of the function $\mathcal{L}$: *formula* $\rightarrow$ *BindingContext* $\rightarrow \wp(Link)$ is the core theoretical contribution of this chapter. $\mathcal{L}$ provides an alternative to $\mathcal{B}$ for interpreting formulae in our constraint language. It expresses results in terms of consistency links rather than true of false.

We will again use a denotational semantics to define $\mathcal{L}$, in order to ensure consistency with our previous definitions of $\mathcal{B}$, and our semantics of XPath. Because the notation is somewhat involved and does not lend itself to straightforward articulation, we will however make use of a further means to aid the discussion.

The language of games, which has been used successfully in the area of game semantics [Hintikka and Sandu, 1996], has turned out to be a very powerful means for supplementing the definition of $\mathcal{L}$. Game semantics have been used to define alternative semantics of first order logic before [Pietarinen, 2000] and are thus suitable also for our purposes.

We will stop short of providing a formal game semantics – our denotational semantics provides the formal framework and basis for proof, — but instead leverage some game-related concepts as metaphors to simplify the discussion. The following is an overview of the vocabulary that we will make reference to, please refer to [Abramsky, 2004] for an excellent and more formal discussion:

- Game semantics treats the evaluation of a formula f against a model M, with a set of free variables s, as a game between two players. In our case, the model is the document set $\mathbb{D}$, and there are no free variables – they are all bound by quantifiers.
- One of the players is the "verifier", denoted by **V**, who wants to prove that a formula holds over $\mathbb{D}$. His opponent is the "falsifier", **F**, who wants to prove the opposite. Whoever succeeds wins the game. The following statements are equivalent: "**V** has a winning strategy for f over $\mathbb{D}$", "the boolean evaluation of f over $\mathbb{D}$ is $\top$".
- When it is a player's turn to move, he decides which action to take according to a "strategy". Informally, this strategy is defined by the function $\mathcal{L}$.
- Where quantifiers lead to variable assignments, the players may use the assignments as "witnesses" to support their case. In our case, the players will provide locator entries into consistency links.
- Game semantics also defines the notions of game composition and role switching for subgames. We will not need to formalise these. We will use role switching for formulas involving negation, where we will simply invert the record of the winner of the subgame, by flipping link statuses.

In the definition of the quantifiers and boolean connectives, we will in fact see that we will go slightly beyond looking for a "winning strategy": we will attempt to find as many winning strategies as possible, always balancing the need to identify the most witnesses with the need not to overwhelm the eventual user with irrelevant information.

With this initial introduction in place, we can now turn to the definition of $\mathcal{L}$. This will be done by showing for each formula construct:

- A formal semantic definition of $\mathcal{L}$ for the construct.
- A **description** that provides a game-oriented interpretation of the modus operandi of $\mathcal{L}$ and clarifies the notation.

- One or more **examples** to demonstrate how the interpretation of the formula handles consistent and inconsistent data, and how it behaves in certain boundary cases.

The specification of the semantics is followed by a discussion of some of the properties of $\mathcal{L}$, such as the applicability of the boolean laws and its relationship to the boolean interpretation $\mathcal{B}$.

The function $\mathcal{L}$ was designed in order to meet practical needs, and later revised to refine the quality of the generated links. During this process, design decisions have been made that have sometimes resulted in a formally slightly less straightforward mapping but have yielded good results. In order to preserve readability, we have delegated discussion of these design decisions and possible alternative definitions to a separate subsection following the definition.

```
<Catalogue>
    <Product>
        <Code>B001</Code>
        <Price currency="GBP">349.95</Price>
    </Product>
    <Product>
        <Code>B002</Code>
        <Price currency="GBP">179.95</Price>
    </Product>
    <Product>
        <Code>B002</Code>
        <Price currency="EUR">50.95</Price>
    </Product>
</Catalogue>
```

Figure 5.2: XML document for constraint examples

We will need to refer to an example document during our definition. The XML document in Figure 5.2 is an abbreviated version of Wilbur's product catalogue. It contains a list of `Product` elements, each with a code, price and currency. When discussing the examples we will refer to the first product element simply as $Product_1$, the second as $Product_2$ and the third as $Product_3$.

The function $\mathcal{L}$

The function $\mathcal{L} : formula \to BindingContext \to \wp(Link)$ takes a formula and a binding context, which will be set to be empty initially, evaluates the formula against the document set $\mathbb{D}$ – the model – and returns a set of consistency links.

In our game interpretation, $\mathcal{L}$ prescribes the steps which the verifier **V** and falsifier **F** should take in order to create a "winning strategy", that is a strategy where the overall boolean outcome of the formula is demonstrated to be $\top$ or $\bot$, respectively. We will use the short notation "win **V**" to record a win for the verifier, and "win **F**" to record a win for the falsifier.

Both the verifier and falsifier will try to assemble "witnesses" along the way, nodes in the documents that support their assertions. The following is an introductory overview of the player's behaviour for some selected constructs:

- $\forall\ v \in p\ (f)$: **F** selects nodes using p and binds them to the variable v in an attempt to make $\mathcal{B}[\![f]\!]$ evaluate to $\bot$ under the binding. If such a node is found, it is recorded as a witness – stored in a link – and **F** wins. Otherwise, **F** concedes and returns no witnesses.
- $\exists\ v \in p\ (f)$: The opposite of the above. **V** gets to choose and wants $\mathcal{B}[\![f]\!]$ evaluate to $\top$ under some assignment. If this is possible, the node bound to v is stored as a witness and **V** wins. Otherwise, **V** concedes and returns no witnesses.
- f_1 *and* f_2: **F** gets to choose, and will choose the formula for which $\mathcal{B}$ evaluates to $\bot$. The witnesses for the evaluation of that subformula are passed on. If both evaluate to $\top$ or both to $\bot$, **F** combines the witnesses and concedes or claims a win, respectively. The other boolean operators will be handled similarly.
- *not* f: **F** and **V** switch roles for the subgame played over f. We execute this switch by flipping the consistency status of links, which essentially turns witnesses produced by one player into witnesses for the other.
- Any predicate, for example $e_1 = e_2$: these are the base cases –**V** wins if $\mathcal{B}$ returns $\top$ for the predicate, otherwise **F** wins. No witnesses are returned.

The definition of $\mathcal{L}$ now begins with the interpretation of the universal quantifier $\forall$:

$$\mathcal{L}[\![\forall v \in p\ (f)]\!]_\rho \quad = \quad \begin{cases} \{(\mathit{Consistent}, [])\},\ \mathcal{B}[\![\forall v \in p\ (f)]\!]_\rho = \top \\ \bigcup_{n_i \in \mathcal{S}_d[\![p]\!]_\rho} ((\mathit{Inconsistent}, [n_i]) \circ \mathcal{L}[\![f]\!]_{bind(v,\{n_i\},\rho)} \mid \\ \qquad \mathcal{B}[\![f]\!]_{bind(v,\{n_i\},\rho)} = \bot),\ \mathit{otherwise} \end{cases}$$

Figure 5.3: Definition of $\mathcal{L}$ for $\forall$

Description The choice of play for $\forall$ is delegated to the falsifier **F**. **F** is trying to find an assignment to the variable v from the nodes selected using $\mathcal{S}_d[\![p]\!]_\rho$ such that the subformula f evaluates to $\bot$ under $\mathcal{B}$.

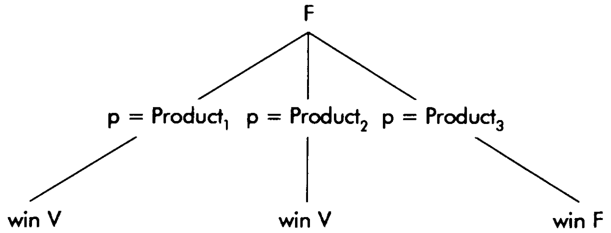
The definition in Figure 5.3 contains a lot of notational detail. The behaviour of $\mathcal{L}$ differs depending on whether the formula evaluates to $\top$ under the boolean interpretation $\mathcal{B}$. If it does, **F** cannot assemble witnesses that support his claim that the formula is violated, and must concede. A consistent link with no entries is recorded and returned as a result. At this stage, we could also generate a link with all nodes that match p, rather than an empty link. This decision will be discussed later in Section 5.2.6 from page 73.

If $\mathcal{B}$ evaluates to $\bot$, then for some assignments to v, $\mathcal{B}[\![f]\!]$ is $\bot$. **F** iterates through these assignments, and uses $\mathcal{L}$ recursively to obtain witnesses in the form of consistency links. He then generates a link to the currently assigned node and combines it with the links produced by the subformula using $\circ$ and the node bound to the variable. **F** returns a set of links, all of which record the alternative witnesses that he has produced in conjunction with the witnesses produced by recursive evaluation.

Example 3. This example checks if all products are priced in GBP:
$\forall\ p \in$/Catalogue/Product($p/Price/@currency='GBP')

Firstly, the path /Catalogue/Product is evaluated and produces the node set $\{\mathit{Product}_1, \mathit{Product}_2, \mathit{Product}_3\}$. For the first two elements, $\mathcal{B}$ returns $\top$, but for the last it returns $\bot$. **F** thus chooses *Product*$_3$ as a witness and produces an inconsistent link $(\mathit{Inconsistent}, [\mathit{Product}_3])$ that is combined with the links returned by $=$. For the sake of the discussion, it is necessary to preempt the semantics of $\mathcal{L}$ in the case of $=$, which is to always return a link with a status reflecting the result of $\mathcal{B}$, and an empty set of locators.

The final result is $\{(\mathit{Inconsistent}, [\mathit{Product}_3])\}$, because joining the link to the empty set of locators returned by $=$ returns a set containing only one link. We can also record this evaluation as a game tree, where internal nodes are labeled with the player who is to move, leaf nodes show who wins if a branch is followed, and the arcs show the chosen variable assignments:

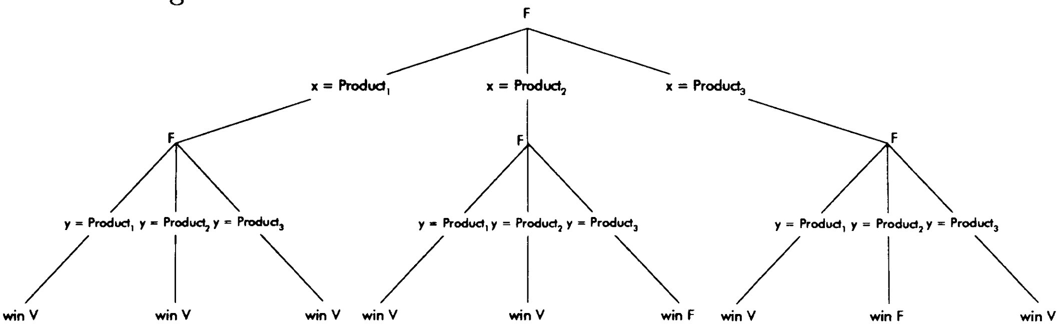


Example 4. We will now take a look at how links are combined using $\circ$ with two nested universal quantifiers. In order to come up with a realistic constraint, we will make use of *implies* and *same*, the semantics of which we will define later: for now suffice it to say that in the inconsistent case the evaluation of *implies* returns an inconsistent link with no locators, due to the definition of the predicates.

Our example formula expresses the constraint that no two products may have the same code:

$$
\begin{array}{l}
\forall x \in \text{/Catalogue/Product(} \\
\quad \forall y \in \text{/Catalogue/Product(} \\
\quad\quad \text{\$x/Code=\$y/Code } \textit{implies same } \$x\ \$y))
\end{array}
$$

Both path expressions evaluate to the same node set, as in the previous example, $\{Product_1, Product_2, Product_3\}$. Below is the game tree for this evaluation of $\mathcal{L}$. It is pruned at the point where the implication occurs, and annotated with who would win if the game were to continue along each branch:


F
x = Product1
x = Product2
x = Product3
F
F
F
y = Product1 y = Product2 y = Product3
y = Product1 y = Product2 y = Product3
y = Product1 y = Product2 y = Product3
win V
win V
win V
win V
win V
win F
win V
win F
win V

There are two branches here that lead to a win for **F** when evaluating the first quantifier. This is because on the second level of these branches – the two right-most ones, – **F** can make a winning choice for at least one child branch.

When x is bound to $Product_2$ and y to $Product_3$ we get an inconsistency as the two nodes are different but have the same code. The set of links returned by $\mathcal{L}$ on the second level, when x is bound to $Product_2$, is $\{(Inconsistent, [Product_3])\}$. This is combined with on the top level with a link to $Product_2$ to form $\{(Inconsistent, [Product_2, Product_3])\}$.

When x is bound to $Product_3$, we get the set of links $\{(Inconsistent, [Product_3, Product_2])\}$. We have thus identified $Product_2$ and $Product_3$ as inconsistent with one another and all other pairs as consistent.

$$
\mathcal{L}[\![\exists v \in p\ (f)]\!]_\rho \;=\; \begin{cases} \{(Inconsistent, [])\},\ \mathcal{B}[\![\exists v \in p\ (f)]\!]_= \bot \\ \bigcup_{n_i \in \mathcal{S}_d[\![p]\!]_\rho} ((Consistent, [n_i]) \circ \mathcal{L}[\![f]\!]_{bind(v,\{n_i\},\rho)} \mid \\ \quad \mathcal{B}[\![f]\!]_{bind(v,\{n_i\},\rho)} = \top),\ otherwise \end{cases}
$$

Figure 5.4: Definition of $\mathcal{L}$ for $\exists$

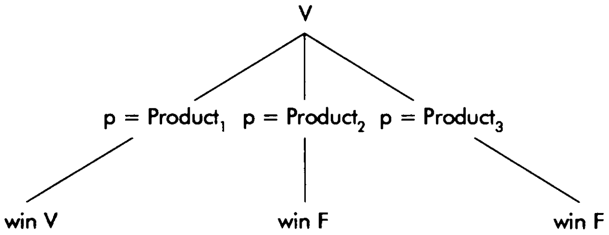
Description For $\exists$, we assign the choice of assignment to the verifier **V**. **V** attempts to find assignments of nodes from $\mathcal{S}_d[\![p]\!]_\rho$ to v, such that $\mathcal{B}$ returns $\top$ for the subformula under the assignment.

If $\mathcal{B}$ returns $\bot$ for the formula as a whole, **V** will not be able to find an assignment and must concede. $\mathcal{L}$ returns an empty consistent link in this case.

If $\mathcal{B}$ is $\top$, then **V** can use $\mathcal{L}$ recursively to gather consistency links that contain witnesses. He generates a link to the currently assigned node and combines it with the set of links returned by the subformula using $\circ$. The set union of all links produced in this way is returned as the result.

This definition for $\exists$ highlights an interesting property of our semantics: the task for **V** and **F** is never to simply find a winning strategy, but to find *all* winning strategies within the evaluation context.

Example 5. Suppose we want to check whether a product with a given code exists anywhere in our document set. This can be expressed as: $\exists\ p \in$/Catalogue/Product($p/Code ='B001'). Only *Product*$_1$ has this code so the game tree looks as follows:



The verifier notes that when v is assigned *Product*$_1$ this makes the boolean result of the child predicate $\top$ and hence leads to a win. $\mathcal{L}$ thus generates a link to *Product*$_1$, which is combined with the empty consistent link returned by evaluating $\mathcal{L}$ on the predicate, to yield $\{(Consistent, [Product_1])\}$.

$\mathcal{B}[\![f_1]\!]_\rho$	$\mathcal{B}[\![f_2]\!]_\rho$	$\mathcal{L}[\![f_1\ and\ f_2]\!]_\rho$
$\top$	$\top$	$\mathcal{L}[\![f_1]\!]_\rho \circ_s \mathcal{L}[\![f_2]\!]_\rho$
$\top$	$\bot$	$\mathcal{L}[\![f_2]\!]_\rho$
$\bot$	$\top$	$\mathcal{L}[\![f_1]\!]_\rho$
$\bot$	$\bot$	$\mathcal{L}[\![f_1]\!]_\rho \circ_s \mathcal{L}[\![f_2]\!]_\rho$

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

Description The definition of $\mathcal{L}$ for the logical connectives *and*, *or* and *implies* is most easily presented in a tabular form. Table 5.1 shows the linking semantics for *and* depending on the boolean results of evaluating the two subformulae f_1 and f_2.

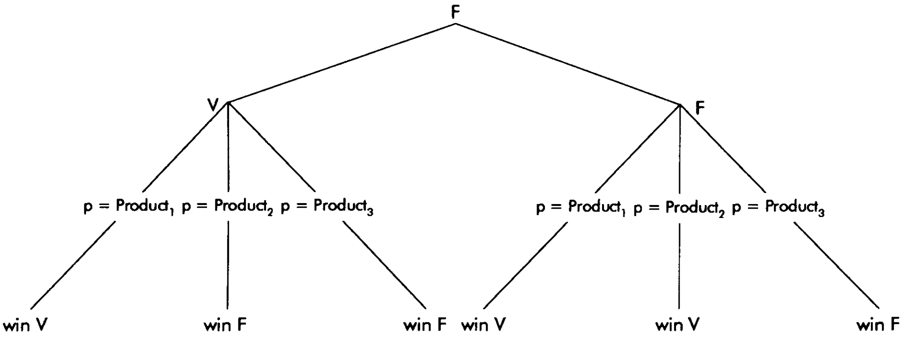
It is the falsifier, **F**, who gets to make the choice in the case of *and*. If only one subformula evaluates to $\bot$ under $\mathcal{B}$, **F** of course picks the links produced by $\mathcal{L}$ for this subformula, since this leads to a win.

If both subformulas evaluate to $\top$ or both evaluate to $\bot$, **F** combines the links produced for the subformulae using $\circ_s$. In a sense, **F** is playing "fair" here because he is producing witnesses – though they are consistent links, witnesses for his opponent – even in the case where there is no winning choice, where both subformulae evaluate to $\top$. This is necessary because the *and* construct may be contained in an outer formula involving negation. In this case, the witnesses would be converted again to being witnesses for **F**.

An alternative way to look at this definition is that the semantics for the logical connectives chooses the links of the subformula that contributes the most *information* to the overall result: a subformula contributes information if changing its boolean result would change the overall boolean result produced by the logical connective. The reasoning behind this approach is that while we want to point out as much information as possible about an inconsistency, at the same time we have to discard as much irrelevant information as possible. If both subformulae evaluate to $\top$, then changing either subformula could make the result $\bot$. Similarly, if both evaluate to $\bot$, changing either has no effect on the overall result. The subformulae contribute an equivalent amount of information, and hence we combine their links.

Example 6. We can use *and* to build up a more complex constraint. For example, we can check that there exists a product with the code B001 and that all products use the currency GBP: (∃ *p* ∈/Catalogue/Product($p/Code ='B001')) *and* (∀ *p* ∈/Catalogue/Product($p/Price/@currency ='GBP')).

We already know from Example 5.2.2, for ∃, that **V** has a winning strategy for the first subformula, where $\mathcal{L}$ returns {(*Consistent*, [*Product*$_1$])}. For the second subformula, *Product*$_3$ has the currency set to EUR, violating the subformula, and is entered into a link as a witness by **F**. The game tree shows this graphically:



At the top level, it is **F**'s turn to choose, and according to the definition of $\mathcal{L}$, he chooses the subformula that evaluates to $\bot$. This is represented by the right branch, where **F** has a winning strategy.

The links produced for the first subformula are thus discarded, as per $\mathcal{L}$, and those produced for the second are returned as the result: {(*Inconsistent*, [*Product*$_3$])}. This matches our intuitive expectations: we do not want to see the links produced for the first subformula because the elements identified in it have not contributed to the inconsistency.

$\mathcal{B}[\![f_1]\!]_\rho$	$\mathcal{B}[\![f_2]\!]_\rho$	$\mathcal{L}[\![f_1 \textit{ or } f_2]\!]_\rho$
$\top$	$\top$	$\mathcal{L}[\![f_1]\!]_\rho \circ_s \mathcal{L}[\![f_2]\!]_\rho$
$\top$	$\bot$	$\mathcal{L}[\![f_1]\!]_\rho$
$\bot$	$\top$	$\mathcal{L}[\![f_2]\!]_\rho$
$\bot$	$\bot$	$\mathcal{L}[\![f_1]\!]_\rho \circ_s \mathcal{L}[\![f_2]\!]_\rho$

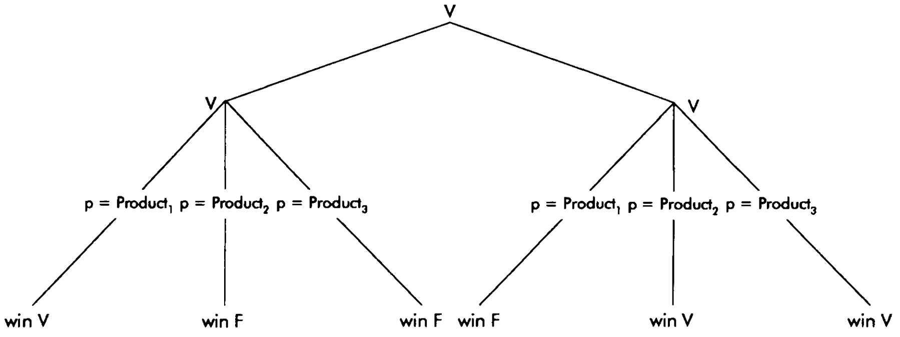
Table 5.2: Definition of $\mathcal{L}$ for *or*

Description The semantics for *or* is quite similar to that for *and*, except that now we point to the subformula that evaluates to $\top$ under $\mathcal{B}$. It is **V**'s turn to play, and **V** of course chooses the option where he has a winning strategy.

If both subformulas evaluate to $\top$, **V** combines the result of applying $\mathcal{L}$ to both using $\circ_s$ – the witnesses in the form of links are combined. If both evaluate to $\bot$, *or* uses a similar "fair" approach to *and* and again combines and forwards the links.

From the alternative, information-driven point of view, if one subformula evaluates to $\bot$ under $\mathcal{B}$, and the other to $\top$, then the latter contains more information: if we changed its result to $\bot$, the formula as a whole would fail. Where both subformulae evaluate to the same result, changing either would not affect the overall result and we combine the links.

Example 7. Suppose we check that a product with code B001 or one with code B002 exists: ($\exists\, p \in$/Catalogue/Product($p/Code ='B001')) *or* ($\exists\, p \in$/Catalogue/Product($p/Code ='B002')). It is **V**'s turn at all steps during the evaluation, leading to the following tree:


V
V
V
p = Product$_1$ p = Product$_2$ p = Product$_3$
p = Product$_1$ p = Product$_2$ p = Product$_3$
win V
win F
win F
win F
win V
win V

Both subformulae evaluate to $\top$, and this is evidenced by the set of links produced for the first, $\{(\mathit{Consistent}, [\mathit{Product}_1])\}$, and the second $\{(\mathit{Consistent}, [\mathit{Product}_2]), (\mathit{Consistent}, [\mathit{Product}_3])\}$.

After combining the sets using $\circ_s$ we get the result set

$$\{(\mathit{Consistent}, [\mathit{Product}_1, \mathit{Product}_2]), (\mathit{Consistent}, [\mathit{Product}_1, \mathit{Product}_3])\}$$

These sets reflect the combination of alternative choices that **V** can make in the branches of the game.

$$\boxed{\mathcal{L}[\![\mathit{not}\ f]\!]_\rho \quad = \quad \mathit{flip}(\mathcal{L}[\![f]\!]_\rho)}$$

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

Description Since the definition of *implies* will entail some negation, we will first discuss *not*. *not* leads to a reversal of the winner/loser situation. Whoever was the winner for f will become the loser for *not* f and vice versa. This is classically represented in game semantics using a "role switch" and a subgame. Since we are using games only for illustration, we will not formalise role switches and subgames. Instead, the definition of $\mathcal{L}$ uses *flip* to change the consistency status of links produced for f. In game terms, any witnesses for **F** then become witnesses for **V** and vice versa.

Example 8. Take again the example constraint $\forall\ p\ \in$/Catalogue/Product($p/Price/ @currency='GBP'). The set of links produced for this was $\{(\mathit{Inconsistent}, [\mathit{Product3}])\}$. In this set, *Product3* is a witness for the violation of the formula.

The set of links for *not* $\forall\ p\ \in$/Catalogue/Product($p/Price/@currency='GBP') is then $\{(\mathit{Consistent}, [\mathit{Product3}])\}$. This makes sense – the formula has been negated and now *Product3* has become a witness for the consistency of the document set against the formula.

$\mathcal{B}[\![f_1]\!]_\rho$	$\mathcal{B}[\![f_2]\!]_\rho$	$\mathcal{L}[\![f_1\ \mathit{implies}\ f_2]\!]_\rho$
$\top$	$\top$	$\mathcal{L}[\![f_2]\!]_\rho$
$\top$	$\bot$	$\mathit{flip}(\mathcal{L}[\![f_1]\!]_\rho \circ_s \mathcal{L}[\![f_2]\!]_\rho)$
$\bot$	$\top$	$\mathit{flip}(\mathcal{L}[\![f_1]\!]_\rho)$
$\bot$	$\bot$	$\mathit{flip}(\mathcal{L}[\![f_1]\!]_\rho)$

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

Description The classical boolean definition of *implies* is such that it returns $\top$, unless the premise – the first subformula – is $\top$ while at the same time the conclusion – the second subformula – is $\bot$. This definition makes *implies* the only asymmetric construct in our language.

Under the classical boolean definition, for any formulas σ and ϕ, $\sigma \rightarrow \phi$ is equivalent to $\neg\sigma \vee \phi$. It is natural then to let **V** choose the strategy for $\mathcal{L}$, similar to the definition of *or*. We will have to think carefully however when it comes to the witnesses that **V** chooses, because the asymmetric definition of *implies* indicates that it is somehow different from *or*.

The first row of the table shows the case where both subformulae evaluate to $\top$ under $\mathcal{B}$. **V** could thus choose the links returned by $\mathcal{L}$ for either subformula as witnesses. However, since we could flip the boolean result for f_1 without affecting the overall result, we regard the witnesses for f_2 as containing more information. **V** thus chooses the links returned for f_2. This is the same result that would be obtained by $\mathcal{L}$ for (*not* f_1) *or* f_2.

The second row is the only row for which the boolean result is $\perp$. Changing the result for either subformula would change the overall result to $\top$, so **V** combines the witnesses in the links. Since f_1 evaluates to $\top$ under $\mathcal{B}$, it will contain consistent links. When the links are combined using $\circ_s$, the overall set will thus contain only consistent links. We therefore use *flip* to make all links in the set inconsistent. Without proof, this is again the same result that would be obtained for (*not* f_1) *or* f_2.

The last two rows are the cases where the premise is $\perp$ under $\mathcal{B}$. In these cases, **V** chooses the witnesses for the premise, and flips their consistency status: by violating $\mathcal{B}$ in the premise, they have become witnesses for **V**. In the case of the third row, this definition marks a departure from the previous boolean equivalence with (*not* f_1) *or* f_2. Under that equivalence, we would have to combine the links for f_1 and f_2, instead of linking to the links produced for the premise. This is a deliberate departure that is designed to reinforce the intuitive rule, *ex falso quodlibet*, – anything follows from an incorrect premise, and the witnesses for the conclusion seem irrelevant. Even if the outcome of the conclusion is altered, the overall result will be unaffected. This slight departure from the boolean interpretation, which has produced good results in our case studies, is a reflection of the increased amount of information our semantics provides, thus amplifying the consequences of defining *implies* as an asymmetric operator.

We will now look at how this definition is applied in a number of examples.

Example 9. Suppose we express the property that if a product exists with code `B001` then a product exists whose price is given in `GBP`: ($\exists$ p $\in$/Catalogue/Product($p/Code ='B001')) *implies* ($\exists$ p $\in$/Catalogue/Product($p/Price/@currency ='GBP')).

Evaluation of the first subformula yields $\{(Consistent, [Product_1])\}$ and of the second $\{(Consistent, [Product_3])\}$. We chose the second set of links as the overall result:

$$\{(Consistent, [Product_3])\}$$

Example 10. If we alter the formula, changing the second subformula to $\forall$, we will end up with an inconsistent result: ($\exists$ p $\in$/Catalogue/Product($p/Code ='B001')) *implies* ($\forall$ p $\in$/Catalogue/Product($p/ Price/@currency ='GBP')). In this case, the conclusion is violated by $Product_3$ and evaluation of the second subformula gives the result $\{(Inconsistent, [Product_3])\}$.

According to the semantics, we thus combine the sets produced by the subformulae to and flip the status of the result to give $\{(\mathit{Inconsistent}, [\mathit{Product}_1, \mathit{Product}_3])\}$. This link expresses that *Product*$_1$ has introduced an inconsistency (by having a node value equal to 'B001') and so has *Product*$_3$ (by having a currency not equal to 'GBP').

Example 11. Finally, we will look at an example where the premise evaluates to $\perp$: (∃ *p* ∈ /Catalogue/Product($p/Code ='unknown')) *implies* (∀ *p* ∈/Catalogue/Product($p/Price/ @currency ='GBP')).

For the first subformula we now get the result $\{(\mathit{Inconsistent}, [])\}$. It does not matter now that one of the products fails the test on the currency since the premise is not fulfilled. Instead, we flip the links from the first subformula and return $\{(\mathit{Consistent}, [])\}$ as our result.

$$\mathcal{L}[\![e_1 = e_2]\!]_\rho = \begin{cases} \{(\mathit{Consistent}, [])\}, & \mathcal{B}[\![e_1 = e_2]\!]_\rho = \top \\ \{(\mathit{Inconsistent}, [])\}, & \mathit{otherwise} \end{cases}$$

Table 5.5: Definition of $\mathcal{L}$ for =

Description The semantics for = is to always return a link with the correct status, but an empty set of locators. This decision was prompted by the following observations:

- Any location path that occurs in = must be relative to a variable introduced by a quantifier in any case – see Definition 4.21 on page 46. Thus it will be possible to unambiguously evaluate $x/Name by following the locator introduced by the quantifier and evaluating the relative path "Name". Any further locators introduced by the equality predicate would be redundant.

- The expressions e_1 and e_2 do not necessarily have to refer to location paths, they could also be numbers or strings. For example, assuming that $x is bound to some node, $x/Name=5 is a valid formula. In cases like these, where an expression does not evaluate to a node set, there are no nodes to link to.

$$\mathcal{L}[\![\mathit{same}\ v_1\ v_2]\!]_\rho = \begin{cases} \{(\mathit{Consistent}, [])\}, & \mathcal{B}[\![\mathit{same}\ v_1\ v_2]\!]_\rho = \top \\ \{(\mathit{Inconsistent}, [])\}, & \mathit{otherwise} \end{cases}$$

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

Description *same* returns a link with no locators for similar reasons to =: the two parameters v_1 and v_2 refer to variables that must have been bound by a quantifier that precedes the predicate. That quantifier would already introduce links to the nodes bound to the variables if necessary, thus any further links would be redundant.

Summary of $\mathcal{L}$

- The function $\mathcal{L}$ evaluates a formula over a document set and returns a set of consistency links.

$$\mathcal{L}\llbracket \forall v \in p\ (f) \rrbracket_\rho \;=\; \begin{cases} \{(\mathit{Consistent}, [])\},\ \mathcal{B}\llbracket \forall v \in p\ (f) \rrbracket_\rho = \top \\ \bigcup_{n_i \in \mathcal{S}_d\llbracket p \rrbracket_\rho} ((\mathit{Inconsistent}, [n_i]) \circ \mathcal{L}\llbracket f \rrbracket_{bind(v,\{n_i\},\rho)} \mid \\ \quad \mathcal{B}\llbracket f \rrbracket_{bind(v,\{n_i\},\rho)} = \bot),\ \mathit{otherwise} \end{cases}$$

$$\mathcal{L}\llbracket \exists v \in p\ (f) \rrbracket_\rho \;=\; \begin{cases} \{(\mathit{Inconsistent}, [])\},\ \mathcal{B}\llbracket \exists v \in p\ (f) \rrbracket_{=}\bot \\ \bigcup_{n_i \in \mathcal{S}_d\llbracket p \rrbracket_\rho} ((\mathit{Consistent}, [n_i]) \circ \mathcal{L}\llbracket f \rrbracket_{bind(v,\{n_i\},\rho)} \mid \\ \quad \mathcal{B}\llbracket f \rrbracket_{bind(v,\{n_i\},\rho)} = \top),\ \mathit{otherwise} \end{cases}$$

$\mathcal{B}\llbracket f_1 \rrbracket_\rho$	$\mathcal{B}\llbracket f_2 \rrbracket_\rho$	$\mathcal{L}\llbracket f_1 \ \mathit{and}\ f_2 \rrbracket_\rho$
$\top$	$\top$	$\mathcal{L}\llbracket f_1 \rrbracket_\rho \circ_s \mathcal{L}\llbracket f_2 \rrbracket_\rho$
$\top$	$\bot$	$\mathcal{L}\llbracket f_2 \rrbracket_\rho$
$\bot$	$\top$	$\mathcal{L}\llbracket f_1 \rrbracket_\rho$
$\bot$	$\bot$	$\mathcal{L}\llbracket f_1 \rrbracket_\rho \circ_s \mathcal{L}\llbracket f_2 \rrbracket_\rho$

$\mathcal{B}\llbracket f_1 \rrbracket_\rho$	$\mathcal{B}\llbracket f_2 \rrbracket_\rho$	$\mathcal{L}\llbracket f_1 \ \mathit{or}\ f_2 \rrbracket_\rho$
$\top$	$\top$	$\mathcal{L}\llbracket f_1 \rrbracket_\rho \circ_s \mathcal{L}\llbracket f_2 \rrbracket_\rho$
$\top$	$\bot$	$\mathcal{L}\llbracket f_1 \rrbracket_\rho$
$\bot$	$\top$	$\mathcal{L}\llbracket f_2 \rrbracket_\rho$
$\bot$	$\bot$	$\mathcal{L}\llbracket f_1 \rrbracket_\rho \circ_s \mathcal{L}\llbracket f_2 \rrbracket_\rho$

$\mathcal{B}\llbracket f_1 \rrbracket_\rho$	$\mathcal{B}\llbracket f_2 \rrbracket_\rho$	$\mathcal{L}\llbracket f_1 \ \mathit{implies}\ f_2 \rrbracket_\rho$
$\top$	$\top$	$\mathcal{L}\llbracket f_2 \rrbracket_\rho$
$\top$	$\bot$	$\mathit{flip}(\mathcal{L}\llbracket f_1 \rrbracket_\rho \circ_s \mathcal{L}\llbracket f_2 \rrbracket_\rho)$
$\bot$	$\top$	$\mathit{flip}(\mathcal{L}\llbracket f_1 \rrbracket_\rho)$
$\bot$	$\bot$	$\mathit{flip}(\mathcal{L}\llbracket f_1 \rrbracket_\rho)$

$$\mathcal{L}\llbracket \mathit{not}\ f \rrbracket_\rho \;=\; \mathit{flip}(\mathcal{L}\llbracket f \rrbracket_\rho)$$

$$\mathcal{L}\llbracket e_1 = e_2 \rrbracket_\rho \;=\; \begin{cases} \{(\mathit{Consistent}, [])\},\ \mathcal{B}\llbracket e_1 = e_2 \rrbracket_\rho = \top \\ \{(\mathit{Inconsistent}, [])\},\ \mathit{otherwise} \end{cases}$$

$$\mathcal{L}\llbracket \mathit{same}\ v_1\ v_2 \rrbracket_\rho \;=\; \begin{cases} \{(\mathit{Consistent}, [])\},\ \mathcal{B}\llbracket \mathit{same}\ v_1\ v_2 \rrbracket_\rho = \top \\ \{(\mathit{Inconsistent}, [])\},\ \mathit{otherwise} \end{cases}$$

5.2.3 On the Relationship of $\mathcal{L}$ and $\mathcal{B}$

It is useful to know what kind of result to expect from the linking semantics when the boolean interpretation of a formula returns $\top$ or $\bot$. Indeed, the recursive definition of $\mathcal{L}$ relies on certain properties: in Table 5.4 on page 63, which gives the definitions of *implies*, the case where the premise is $\top$ and the conclusion $\bot$ relies on the first subformula returning only consistent links. It then uses *flip* in the expectation of returning a set of inconsistent links. This expectation is supported by the following two theorems.

Theorem 5.1. *For any formula f, $\mathcal{B}[\![f]\!] = \top$ if and only if $\mathcal{L}[\![f]\!]$ contains only consistent links.*

Theorem 5.2. *For any formula f, $\mathcal{B}[\![f]\!] = \bot$ if and only if $\mathcal{L}[\![f]\!]$ contains only inconsistent links.*

Proof. Both theorems can be shown to hold using structural induction over the abstract syntax of the formula language. Since they are interdependent, we will prove both at the same time using mutual induction. We will first show that $\mathcal{L}$ follows from $\mathcal{B}$, and then the reverse.

Consider the implications, if $\mathcal{B}[\![f]\!] = \top$ then $\mathcal{L}[\![f]\!]$ contains only consistent links, and if $\mathcal{B}[\![f]\!] = \bot$ then $\mathcal{L}[\![f]\!]$ contains only inconsistent links. Our bases cases are the definitions of $=$ and *same* as they are non-recursive. Both are defined to return a link with a status reflecting the boolean result directly, and thus both theorems hold. For the following recursive cases, consisting of the quantifiers, the boolean connectives and *not*, we will use the induction hypothesis that the theorems hold for the subformulas and show they hold after $\mathcal{L}$ is applied to the whole formula.

Consider the quantifiers: If $\mathcal{B}[\![\forall\ v \in p\ (f)]\!] = \top$ then $\mathcal{L}[\![\forall\ v \in p\ (f)]\!] = \{(Consistent, [])\}$. If $\mathcal{B}[\![\forall\ v \in p\ (f)]\!] = \bot$ then there is at least one assignment to the variable for which an inconsistent link will be generated. For $\exists$, if $\mathcal{B}[\![\exists\ v \in p\ (f)]\!] = \top$ then for at least one assignment to the variable a consistent link will be generated. If $\mathcal{B}[\![\exists\ v \in p\ (f)]\!] = \bot$ then $\mathcal{L}[\![\exists\ v \in p\ (f)]\!] = \{(Inconsistent, [])\}$. Therefore, if the induction hypothesis holds for the subformulae, it will hold for the quantifiers. Now consider the logical connectives.

For *and*, if $\mathcal{B}[\![f_1\ and\ f_2]\!] = \top$ then $\mathcal{B}[\![f_1]\!] = \top$ and $\mathcal{B}[\![f_2]\!] = \top$. Under the induction hypothesis, the set of links returned for f_1 and f_2 contains only consistent links. We use $\circ_s$ to combine the two sets. The definition of $\circ_s$ assigns each link in the result set a status from a link in the first set. By induction, therefore, all links in the result set will be consistent. If $\mathcal{B}[\![f_1\ and\ f_2]\!] = \bot$, we have three cases to consider: exactly one subformula evaluates to $\bot$ or both evaluate to $\bot$. If only one subformula evaluates to $\bot$, the definition of $\mathcal{L}$ returns the links for that subformula. Under the induction hypothesis, these links are all inconsistent. If both subformulae evaluate to $\bot$, under the induction hypothesis they return either empty sets or sets of inconsistent links. From the definition of $\circ_s$ it again follows that the result contains only inconsistent links. Therefore, by induction, the theorems hold for f_1 *and* f_2.

The definition of *or* mirrors the definition of *and* and hence the proof is a corollary. Assume then that $\mathcal{B}[\![f_1 \textit{ implies } f_2]\!] = \top$. This means that either $\mathcal{B}[\![f_1]\!] = \top$ and $\mathcal{B}[\![f_2]\!] = \top$ or $\mathcal{B}[\![f_1]\!] = \bot$ and f_2 evaluates to either result. In the first case, the definition of $\mathcal{L}$ returns $\mathcal{L}[\![f_2]\!]$, which, under the induction hypothesis, contains only consistent links. In the second case, we take the links produced by $\mathcal{L}[\![f_1]\!]$, which are all inconsistent under the induction hypothesis, and flip their status. They will thus all be consistent. Now consider the case where $\mathcal{B}[\![f_1 \textit{ implies } f_2]\!] = \bot$. Then $\mathcal{B}[\![f_1]\!] = \top$ and $\mathcal{B}[\![f_2]\!] = \bot$. The definition of $\mathcal{L}$ uses $\circ_s$ to combine the links returned for f_1, which are all consistent under the induction hypothesis, with those returned for f_2, which are all inconsistent under the induction hypothesis. Since $\circ_s$ assigns the status of the links in the first set to all resulting links, all links returned by the function will be consistent. The definition of $\mathcal{L}$ then applies $flip$ to the set of links and hence all resulting links will be inconsistent. By induction, the theorems hold for f_1 *implies* f_2.

The remaining case is *not*. The definition of *not* flips the status of all links produced by the subformula, so the result follows trivially under the induction hypothesis.

We will now look at the converse statements, "if $\mathcal{L}[\![f]\!]$ contains only consistent links then $\mathcal{B}[\![f]\!] = \top$" and "if $\mathcal{L}[\![f]\!]$ contains only inconsistent links then $\mathcal{B}[\![f]\!] = \bot$". We will again use mutual induction, starting with the predicate base cases.

The base cases are trivial from the definition. If a set with a consistent link is generated, it is because the boolean result was $\top$ and vice versa for inconsistent links.

The quantifier cases are also trivial: for $\forall$ the definition shows that sets containing consistent links are generated if the boolean result is true, and vice versa for inconsistent links. Note that the definition of $\circ_s$ assigns the status *Inconsistent*, of the link produced by the quantifier, to all links in the result set. The definition of $\exists$ is along similar lines.

Now consider *and* - we can show that the theorem holds by reductio ad absurdum: suppose $\mathcal{L}[\![f_1 \textit{ and } f_2]\!]$ contains only consistent links, but at the same time $\mathcal{B}[\![f_1 \textit{ and } f_2]\!] = \bot$. The latter is the case if exactly one of the subformulae returns $\bot$, or both return $\bot$. Assume $\mathcal{B}[\![f_1]\!] = \bot$ and $\mathcal{B}[\![f_2]\!] = \top$. From the definition of *and*, we can see that we return $\mathcal{L}[\![f_1]\!]$ as the result - but through the induction hypothesis we know that all links in $\mathcal{L}[\![f_1]\!]$ are inconsistent because the boolean result is $\bot$, and similarly for the case where $\mathcal{B}[\![f_1]\!] = \top$ and $\mathcal{B}[\![f_2]\!] = \bot$. Now consider the case where $\mathcal{B}[\![f_1]\!] = \bot$ and $\mathcal{B}[\![f_2]\!] = \bot$. The induction hypothesis tells us that both $\mathcal{L}[\![f_1]\!]$ and $\mathcal{L}[\![f_2]\!]$ return only inconsistent links, and through the definition of $\circ_s$ we know that the overall result contains only inconsistent links. We have thus failed to find a case where $\mathcal{L}[\![f_1 \textit{ and } f_2]\!]$ contains consistent links and $\mathcal{B}[\![f_1 \textit{ and } f_2]\!] = \bot$. To prove the reverse case, assume that $\mathcal{L}[\![f_1 \textit{ and } f_2]\!]$ contains only inconsistent links and at the same time $\mathcal{B}[\![f_1 \textit{ and } f_2]\!] = \top$. The boolean result will only be $\top$ if both subformulae evaluate to $\top$. In this case, the definition of *and* joins the links produced by the subformulae, which contain only consistent links by the induction hypothesis - hence the overall result will only contain consistent links.

The proof for the disjunctive *or* mirrors that of *and*, so we will look at *implies*. Assume $\mathcal{L}[\![f_1 \textit{ implies } f_2]\!]$ contains only consistent links, but at the same time $\mathcal{B}[\![f_1 \textit{ implies } f_2]\!] = \bot$. From the boolean definition we infer that the latter can only be the case if $\mathcal{B}[\![f_1]\!] = \top$ and $\mathcal{B}[\![f_2]\!] = \bot$. In this case we would return $\mathit{flip}(\mathcal{L}[\![f_1]\!] \circ_s \mathcal{L}[\![f_2]\!])$. Our induction hypothesis tells us that the links in $\mathcal{L}[\![f_1]\!]$ are all consistent - hence through the definition of $\circ_s$ and flip, all links in the result set are inconsistent and our assumption is rejected. Consider the reverse case where $\mathcal{L}[\![f_1 \textit{ implies } f_2]\!]$ contains only inconsistent links, but $\mathcal{B}[\![f_1 \textit{ implies } f_2]\!] = \top$. There are three cases to consider: if $\mathcal{B}[\![f_1]\!] = \top$ and $\mathcal{B}[\![f_2]\!] = \top$ we return $\mathcal{L}[\![f_2]\!]$, which under the induction hypothesis contains only consistent links. If $\mathcal{B}[\![f_1]\!] = \bot$, and $\mathcal{B}[\![f_2]\!]$ is either $\top$ or $\bot$, we return $\mathit{flip}(\mathcal{L}[\![f_1]\!])$. Since $\mathcal{L}[\![f_1]\!]$ contains only inconsistent links under the induction hypothesis, and flip inverts their status, we again obtain only consistent links. The result holds by reductio ad absurdum.

Finally, consider *not*. Assume a case where $\mathcal{L}[\![\textit{not } f]\!]$ contains only consistent links and $\mathcal{B}[\![\textit{not } f]\!] = \bot$. Then, by the boolean definition $\mathcal{B}[\![f]\!] = \top$ and by the induction hypothesis, $\mathcal{L}[\![f]\!]$ contains only consistent links. flip inverts the links to make them all inconsistent – the result holds by reductio ad absurdum. The result for the case of inconsistent links is analogous. □

5.2.4 A Special Case

For practical, as well as historical reasons with the way that the formula language has evolved from a simple constraint between two sets, to more complex boolean constraints between two sets, to its current first order logic form, our implementation assigns a different semantics to $\mathcal{L}[\![\forall\ v \in p(f)]\!]$ where the $\forall$ is the outermost formula construct. We point this out here to ensure consistency of the results reported in the case study with the semantic definition.

Whereas the previous definition generates an inconsistent link to nodes under whose assignment to the variable the subformula returns $\bot$, the new definition also generates consistent links when the subformula evaluates to $\top$.

The constraint $\forall$ p ∈/Catalogue/Product($p/Price/@currency='GBP'), shown as an example in the definition of $\forall$, would then evaluate to $\{(\mathit{Consistent}, [\mathit{Product}_1]), (\mathit{Consistent}, [\mathit{Product}_2]), (\mathit{Inconsistent}, [\mathit{Product}_3])\}$. This gives us the ability to identify both consistent and inconsistent data items at the same time.

We have introduced this special semantics for two reasons:

- Because an earlier incarnation of the semantics restricted formulae to start with $\forall$, with a similar distinction between universal quantifiers that occur as the outermost construct or on the inside, and this semantics has produced good results in our case studies.
- Because many of the constraint we have expressed start with $\forall$, as a means of choosing an element as a context for the subformula. For example, one might say "for all classes, a certain constraint holds" or "for all products, a certain constraint holds". In these cases, we can provide valuable extra information by identifying both the consistent and inconsistent nodes.

5.2.5 Some Properties

This section shows that our linking function $\mathcal{L}$ has properties that are similar to the usual boolean laws, including relationships between the quantifiers and de Morgan's laws. We retain some of the intuitive character of the standard boolean semantics.

Theorem 5.3. $\mathcal{L}[\![\forall\ v \in p\ (f)]\!]_\rho = \mathcal{L}[\![not(\exists\ v \in p\ (not\ f))]\!]_\rho$

Proof. Suppose $\mathcal{B}[\![\forall\ v \in p\ (f)]\!]_\rho = \top$.

1. $\mathcal{L}[\![\forall\ v \in p\ (f)]\!]_\rho = \{(Consistent, [])\}$ by the definition of $\mathcal{L}$ for $\forall$.
2. $\mathcal{L}[\![not(\exists\ v \in p\ (not\ f))]\!]_\rho = flip(\mathcal{L}[\![\exists\ v \in p\ (not\ f)]\!]_\rho)$ by the definition of $\mathcal{L}$ for *not*.
3. Since $\mathcal{B}[\![\forall\ v \in p\ (f)]\!]_\rho = \top$, f evaluates to $\top$ for all assignments to v.
4. Therefore, *not* f evaluates $\perp$ for all assignments to v.
5. From 2, 4, and the definitions of $\mathcal{L}$ for $\exists$ and *flip*: $flip(\mathcal{L}[\![\exists\ v \in p\ (not\ f)]\!]_\rho) = flip(\{(Inconsistent, [])\}) = \{(Consistent, [])\}$.
6. The result in 1 equals the result in 5.

Now suppose $\mathcal{B}[\![\forall\ v \in p\ (f)]\!]_\rho = \perp$.

1. Some nodes $n_1, ..., n_n$ cause the subformulae f to evaluate to $\perp$.
2. $\mathcal{L}[\![\forall\ v \in p\ (f)]\!]_\rho = \bigcup_{n_i}(Inconsistent, [n_i]) \circ \mathcal{L}[\![f]\!]_{bind(v,\{n_i\},\rho)}$, from 1 and the definition of $\forall$.
3. The same set of nodes $n_1, ..., n_n$ from 1 causes *not* f to evaluate to $\top$.

4. $\mathcal{L}[\![not(\exists\ v \in p\ (not\ f))]\!]_\rho = flip(\mathcal{L}[\![\exists\ v \in p\ (not\ f)]\!]_\rho)$, from the definition of $\mathcal{L}$ for *not*.

5. $flip(\mathcal{L}[\![\exists\ v \in p\ (not\ f)]\!]_\rho) = flip(\bigcup_{n_i}(Consistent, [n_i]) \circ \mathcal{L}[\![f]\!]_{bind(v,\{n_i\},\rho)})$, from 3, 4 and the definition of $\exists$.

6. $flip((s,l) \circ links) = flip(\{(s, l+l_i) \mid (s_i, l_i) \in links\})$, for any status s, list of locators l and set of links $links$, from the definition of $\circ$.

7. $flip(\{(s, l+l_i) \mid (s_i, l_i) \in links\}) = \{fliplink((s, l+l_i)) \mid (s_i, l_i) \in links\}$, from 6 and the definition of $flip$ ($fliplink$ is locally defined in the definition of $flip$).

8. $flip(\bigcup_{n_i}(Consistent, [n_i]) \circ \mathcal{L}[\![f]\!]_{bind(v,\{n_i\},\rho)}) = flip(\bigcup_{n_i}\{(Consistent, [n_i] + l_j) \mid (s_j, l_j) \in \mathcal{L}[\![f]\!]_{bind(v,\{n_i\},\rho)}\})$, from 5 and 6.

9. $flip(\bigcup_{n_i}\{(Consistent, [n_i] + l_j]) \mid (s_j, l_j) \in \mathcal{L}[\![f]\!]_{bind(v,\{n_i\},\rho)})\}) =$
$\bigcup_{n_i}\{fliplink((Consistent, [n_i] + l_j])) \mid (s_j, l_j) \in \mathcal{L}[\![f]\!]_{bind(v,\{n_i\},\rho)})\} =$
$\bigcup_{n_i}\{(Inconsistent, [n_i] + l_j]) \mid (s_j, l_j) \in \mathcal{L}[\![f]\!]_{bind(v,\{n_i\},\rho)})\}$, from 7, 8 and the definition of $fliplink$.

10. $\bigcup_{n_i}\{(Inconsistent, [n_i] + l_j) \mid (s_j, l_j) \in \mathcal{L}[\![f]\!]_{bind(v,\{n_i\},\rho)})\}=$
$\bigcup_{n_i}(Inconsistent, [n_i]) \circ \mathcal{L}[\![f]\!]_{bind(v,\{n_i\},\rho)}$, from the 9 and definition of $\circ$.

11. The result in 2 is equal to the result in 10.

□

Theorem 5.4. $\mathcal{L}[\![\exists\ v \in p\ (f)]\!] = \mathcal{L}[\![not(\forall\ v \in p\ (not\ f))]\!]$

Proof. Omitted as it proceeds along exactly the same lines as the previous proof. □

Theorem 5.5. $\mathcal{L}[\![not(not\ f)]\!] = \mathcal{L}[\![f]\!]$

Proof. Let $\mathcal{L}[\![f]\!] = l$. Then $\mathcal{L}[\![not\ f]\!] = flip(l)$ and $\mathcal{L}[\![not(not\ f)]\!] = flip(flip(l))$. $flip$ does not change the number of links in l and does not affect the locators of any links. For any link, it changes the status from consistent to inconsistent, or vice versa. The result follows. □

Lemma 5.13. $flip(l_1 \circ_s l_2) = flip(l_1) \circ_s flip(l_2)$ *if all links in* l_1 *and* l_2 *have the same status.*

Proof. Consider the boundary cases: $flip(l \circ_s \emptyset) = flip(l) = flip(l) \circ_s flip(\emptyset)$ and $flip(\emptyset \circ_s l) = flip(l) = flip(\emptyset) \circ_s flip(l)$, from the definition of $\circ_s$.

Now take any non-empty sets of links l_1 and l_2. Assume all links in l_1 and l_2 have the status s. Then all links in $l_1 \circ_s l_2$ have the same status s by the definition of $\circ_s$. Therefore, all links in $flip(l_1 \circ_s l_2)$ have the opposite status, $\bar{s}$. All links in $flip(l_1)$ have the status $\bar{s}$ and so do all links in $flip(l_2)$. Therefore all links in $flip(l_1) \circ_s flip(l_2)$ have the status $\bar{s}$. Furthermore, the links in $l_1 \circ_s l_2$ have the same locators as those in $flip(l_1) \circ_s flip(l_2)$ as the concatenation of the locators does not depend on the links' status. Therefore, the links in $flip(l_1 \circ_s l_2)$ have the same locators and status as the links in $flip(l_1) \circ_s flip(l_2)$ and hence the sets of links are equivalent. □

Theorem 5.6. *de Morgan's law:* $\mathcal{L}[\![not(f_1\ and\ f_2)]\!] = \mathcal{L}[\![(not\ f_1)\ or\ (not\ f_2)]\!]$

Proof. Consider the four cases:

- $\mathcal{B}[\![f_1]\!] = \top$ and $\mathcal{B}[\![f_2]\!] = \top$: $\mathcal{L}[\![not(f_1\ and\ f_2)]\!] = flip(\mathcal{L}[\![f_1]\!] \circ_s \mathcal{L}[\![f_2]\!])$. By Lemma 5.13, and Theorem 5.1, which guarantees the two link sets contain links with the same status, $flip(\mathcal{L}[\![f_1]\!] \circ_s \mathcal{L}[\![f_2]\!]) = flip(\mathcal{L}[\![f_1]\!]) \circ_s flip(\mathcal{L}[\![f_2]\!])$.

  Since $\mathcal{B}[\![f_1]\!] = \top$, $\mathcal{B}[\![not\ f_1]\!] = \bot$ and hence $\mathcal{L}[\![(not\ f_1)\ or\ (not\ f_2)]\!] = \mathcal{L}[\![(not\ f_1)]\!] \circ_s \mathcal{L}[\![(not\ f_2)]\!]$ by the definition of *or*. $\mathcal{L}[\![(not\ f_1)]\!] \circ_s \mathcal{L}[\![(not\ f_2)]\!] = flip(\mathcal{L}[\![f_1]\!]) \circ_s flip(\mathcal{L}[\![f_2]\!])$, equivalent to *and*.

- $\mathcal{B}[\![f_1]\!] = \top$ and $\mathcal{B}[\![f_2]\!] = \bot$: $\mathcal{L}[\![not(f_1\ and\ f_2)]\!] = flip(\mathcal{L}[\![f_1\ and\ f_2]\!]) = flip(\mathcal{L}[\![f_2]\!]_)$ from the definition of *and*. $\mathcal{L}[\![(not\ f_1)\ or\ (not\ f_2)]\!] = \mathcal{L}[\![not\ f_2]\!]$ from the definition of *or*, and $\mathcal{L}[\![not\ f_2]\!] = flip(\mathcal{L}[\![f_2]\!])$ as before.

- $\mathcal{B}[\![f_1]\!] = \bot$ and $\mathcal{B}[\![f_2]\!] = \top$: corollary to previous case.

- $\mathcal{B}[\![f_1]\!] = \bot$ and $\mathcal{B}[\![f_2]\!] = \bot$: corollary to first case.

□

Theorem 5.7. *de Morgan's law:* $\mathcal{L}[\![not(f_1\ or\ f_2)]\!] = \mathcal{L}[\![(not\ f_1)\ and\ (not\ f_2)]\!]$

Proof. Again consider four cases:

- $\mathcal{B}[\![f_1]\!] = \top$ and $\mathcal{B}[\![f_2]\!] = \top$: $\mathcal{L}[\![not(f_1\ or\ f_2)]\!] = flip(\mathcal{L}[\![f_1]\!] \circ_s \mathcal{L}[\![f_2]\!])$. By Lemma 5.13, and Theorem 5.1, $flip(\mathcal{L}[\![f_1]\!] \circ_s \mathcal{L}[\![f_2]\!]) = flip(\mathcal{L}[\![f_1]\!]) \circ_s flip(\mathcal{L}[\![f_2]\!])$.

  Since $\mathcal{B}[\![f_1]\!] = \top$, $\mathcal{B}[\![not\ f_1]\!] = \bot$ and hence $\mathcal{L}[\![(not\ f_1)\ and\ (not\ f_2)]\!] = \mathcal{L}[\![(not\ f_1)]\!] \circ_s \mathcal{L}[\![(not\ f_2)]\!]$ by the definition of *and*. $\mathcal{L}[\![(not\ f_1)]\!] \circ_s \mathcal{L}[\![(not\ f_2)]\!] = flip(\mathcal{L}[\![f_1]\!]) \circ_s flip(\mathcal{L}[\![f_2]\!])$, equivalent to *or*.

- $\mathcal{B}[\![f_1]\!] = \top$ and $\mathcal{B}[\![f_2]\!] = \bot$: $\mathcal{L}[\![not(f_1\ or\ f_2)]\!] = flip(\mathcal{L}[\![f_1\ or\ f_2]\!]) = flip(\mathcal{L}[\![f_1]\!]_)$ from the definition of *or*. $\mathcal{L}[\![(not\ f_1)\ and\ (not\ f_2)]\!] = \mathcal{L}[\![not\ f_1]\!]$ from the definition of *and*, and $\mathcal{L}[\![not\ f_1]\!] = flip(\mathcal{L}[\![f_1]\!])$ as before.

- $\mathcal{B}[\![f_1]\!] = \bot$ and $\mathcal{B}[\![f_2]\!] = \top$: corollary to previous case.

- $\mathcal{B}[\![f_1]\!] = \bot$ and $\mathcal{B}[\![f_2]\!] = \bot$: corollary to first case.

□

5.2.6 Discussion and Alternatives in the Semantic Definition

We have considered some alternatives for our semantic definition that should provide useful background about its current state. Section 5.2.4 has already shown that some amount of practical experience has been fed back into the design of the language and the following discussion is also motivated by practical concerns.

In the definition of $\mathcal{L}$ for $\exists$, we link to nodes for which the given subformula evaluates to $\top$. If we cannot find any such nodes, we produce a link of the form $(\mathit{Inconsistent}, [])$ to signify that no node could be found that made the subformula true. This is not the only choice at this point and some alternatives were considered. Take the example formula $\exists\ p \in$/Catalogue/Product(Code="unknown"). We have four choices:

1. Provide a link of the form $(\mathit{Inconsistent}, [])$ as in our semantics.
2. Create a link of the form $(\mathit{Inconsistent}, [\mathit{Product}_1, \mathit{Product}_2, \mathit{Product}_3])$ to list the nodes that were considered as part of the check.
3. Create several links, $(\mathit{Inconsistent}, [\mathit{Product}_1])$, $(\mathit{Inconsistent}, [\mathit{Product}_2])$ and $(\mathit{Inconsistent}, [\mathit{Product}_3])$.
4. Create a link that identifies the set of nodes we have checked using some identifier, perhaps the path: $(\mathit{Inconsistent}, /\mathit{Catalogue}/\mathit{Product})$.

Each of these approaches has advantages and disadvantages. The first, and the one eventually chosen, is simple and intuitive, and makes the semantics easy to define. As a downside, it does not always return the maximum amount of information because it throws away any links generated by subformulae as irrelevant. Consider a formula of the form, "there exists a class where all attributes start with an x". Now suppose we cannot find such a class. Then for all classes, at least one attribute will not have started with an x. Our semantics will not link to these inconsistent attributes but will, by virtue of producing a single inconsistent link, record the fact that the constraint was false for all classes.

Now consider the second approach. According to our semantics, in the constraint from the previous paragraph the locators pointing to the inconsistent attributes will be combined with the single locator that points to all nodes. It would be preferable to combine only the locator pointing to a single class with those pointing to its attributes. This problem aside, storing all locators in a single link also makes the number of locators a link carries dependent on the number of elements matched in the documents rather than the number of nested quantifiers. This makes it impossible to predict in advance the types of links that will be generated. This in turn makes static analysis hard, and makes it extremely difficult to use the links for follow-on tasks like repair generation.

The third approach overcomes the limitation on static analysis that the second suffers. It also yields good links with the example rule: each class is combined with the attributes that need to be changed to remove the inconsistency. The final, fourth approach, was rejected as too complex as it would make the concept of a locator polymorphic.

The motivation to choose the first approach over the third, which seems to offer more diagnostic value with little additional complexity stems from practical considerations. Firstly, by far the majority of rules we have seen in practice are simple and do not encounter the problem discussed here. Secondly, it is not clear that it is necessarily intuitive to call a node inconsistent because it fails the existential quantifier's test: when we say "there exists a node with a certain property x", and we cannot find such a node, it does not necessarily make sense to call all nodes that we have tested inconsistent – it may be that the inconsistency is caused by the absence of a new node. And thirdly, by linking to all nodes that are inconsistent we introduce a fairly large number of links. Suppose we want to test that for all entries in a document, there exists another entry in a set of thousands of documents, such that their identifiers match. For every entry in the document that does not have a match we would create thousands of links rather than just a link identifying the entry.

It was mainly the first reason, the structure of the majority of the rules we have seen in practice, and the third, preventing an explosion in the number of links, that have led us to choose the first approach.

It has also been pointed out to us that our approach is a departure from the intuitive definition of universal and existential quantification as a conjunction or disjunction, respectively. For example, given the formula $\forall\ p\ \in$/Catalogue/Product(f), and three elements $Product_1$, $Product_2$, $Product_3$, we may expect the result of evaluation to be the same as that of $\mathcal{L}[\![f]\!]_{Product_1}$ *and* $\mathcal{L}[\![f]\!]_{Product_2}$ *and* $\mathcal{L}[\![f]\!]_{Product_3}$, using a shortcut notation for binding the product elements to p. This equivalence does not hold – the definition of $\mathcal{L}$ for $\forall$ is to return a consistent link with no locators in the consistent case, whereas the conjunction would combine the links produced by subformulae. In the inconsistent case, the existing definition would create a link to the violating $Product_i$, whereas the conjunction would again only combine the link created by the subformulae – since only quantifiers introduce locators, we would be losing a witness. Since quantifiers are usually used to iterate over larger sets of nodes and it is rare to find an *and* combining more than two elements of the same type, we believe that this inequality does not introduce any problems in practice. We are also keen to retain the feature that only quantifiers, which are explicit iteration constructs that assign bindings to variables, should introduce locators. This is intuitive for constraint writers and enables the specification of reporting components that can refer to locators using variable references – a feature that would otherwise be hard to achieve if boolean connectives introduced locators, since we would have to identify break locations using paths through the game trees instead.

We have also considered an alternative in the definition of the logical connectives *and* and *or*. These connectives combine the link sets of their subformulae using $\circ_s$ if the boolean result of evaluating the subformulae is equivalent, that is both evaluate to $\top$ or both evaluate to $\bot$. It would have also been possible to simply return the union of the result sets of the overall result. This difference is best illustrated using an example.

Suppose we evaluated the formula ($\forall\, p\in$/Catalogue/Product($p/Price/@currency='GBP')) *or* (($\forall\, p\in$/Catalogue/Product($p/Code='B002')). This formula evaluates to $\bot$ and our linking semantics would yield the set of links $\{(\mathit{Inconsistent}, [\mathit{Product}_3, \mathit{Product}_1])\}$. If we redefined it to use set union instead of $\circ_s$ we would get $\{(\mathit{Inconsistent}, [\mathit{Product}_3]), (\mathit{Inconsistent}, [\mathit{Product}_1])\}$.

This raises two questions: what does it really mean for nodes to be included in a set of locators, and which of the representation above make more sense? Suppose we take the first link to read "$\mathit{Product}_3$ is inconsistent with $\mathit{Product}_1$". If we take the inclusion of the nodes in the same set of locators to mean that they are consistent with one another, we seem to contravene the intention of the constraint: the constraint writer wanted one or the other to hold independently but we have somehow established a connection between them.

It seems to make more sense to interpret the locators to mean that $\mathit{Product}_3$ and $\mathit{Product}_1$ simultaneously contribute to an inconsistency, but not necessarily because of a relationship between each other. The important thing is that the list of locators contains all the items that have been associated with an inconsistency so a choice can be made which one to alter if necessary.

Now consider the result we would get if we chose union instead of $\circ_s$ for our semantics, $\{(\mathit{Inconsistent}, [\mathit{Product}_3]), (\mathit{Inconsistent}, [\mathit{Product}_1])\}$. One could argue that this result captures the independent contribution of the two elements to the overall inconsistency better. However, if we now change one of $\mathit{Product}_1$ or $\mathit{Product}_3$, we would cause both links to disappear because the formula would become $\top$. It seemed preferable that *all* the information whose modification would have an impact on the status of a link should be contained in the link itself, and that dependencies between links should be avoided if possible.

5.3 Chapter Summary

In this chapter we have provided two evaluation semantics for the constraint language specified in the last chapter. The evaluation semantics specify how constraints are evaluated over multiple DOM trees to provide inconsistency reports.

We have seen that the standard boolean interpretation of our constraint language is not rich enough to provide diagnostics that help us to track inconsistency between multiple documents. Our novel consistency link semantics overcomes this by accumulating information during evaluation that effectively records which elements have acted as "witnesses" to constraint violations.

The relationship between the boolean and link generation semantics has been established by showing that a number of common properties that hold for the former still hold for the latter. In particular, we have seen that if a boolean evaluation of a constraint is *true*, we expect to see consistent links, and vice versa for the inconsistent case. Our linking semantics thus acts as expected from languages based on first order logic, while at the same time providing the additional level of detail in diagnostic information that we have been seeking.

The following chapter will deal with more practical issues that arise with the implementation of the semantics as part of a consistency checking service.

6 Implementation

We have implemented our link generation semantics in a consistency check engine. The engine was first prototyped in Haskell to clarify implementation details in the semantics, and then implemented in Java to deal with actual input data.

The implementation required some additional mechanisms and design decisions: constraints have to be presented in some concrete syntax and fed into the check engine, and results have to be returned. We will look at examples demonstrating XML representations of our constraint language and linkbase outputs. Documents and the constraints that are to be applied to them have to be managed: we will look at our document set and rule set mechanisms, as well as "fetchers", which are used to translate non-XML data into an XML format for checking.

Our check engine is encapsulated in a single component and can thus be deployed in a number of architectures. We will take a look at a web based architecture, as well as a message queuing architecture that checks documents as part of a message flow. These examples are mainly intended to show the adaptability of a properly encapsulated, lightweight consistency checking service.

Finally, we will concern ourselves with a practical presentation of the checking output. Consistency links between elements represent, in a sense, the *raw* diagnostic output of the check engine, but are not always readily accessible to the user. They can, however, be used to visualise relationships between documents, to automatically provide consistency reports that use link locators to display information from documents, or to drive consistency reporting directly in document manipulation tools. We will look at several practical processing components we have built that make use of consistency links.

6.1 XML Constraints and Links

We use XML as a concrete syntax for our language and the resulting linkbase. Encoding the language in XML has a number of advantages: it is easy to parse using off the shelf tools, it blends more uniformly into the environment since all other documents are assumed to be in XML or transformable into XML, and it enables us to use related technologies like XSLT [Clark, 1999] to transform the constraints for presentation to the user.

A presentation of the entire language in XML would not add significantly to the discussion. Instead, the grammar is given in Appendix B and we will look at a constraint from the running example. Figure 6.1 shows the now familiar constraint *"for all Advert elements, there exists a Product element in the Catalogue such that their names match"* in XML.

```
<consistencyrule id="r1">
   <header>
     <description>
       Each advert must refer to a product
       defined in the catalogue
     </description>
   </header>

   <forall var="a" in="/Adverts">
       <exists var="p" in="/Catalogue/Products">
           <equal op1="$a/ProductName/text()"
                  op2="$p/Name/text()"/>
       </exists>
   </forall>
</consistencyrule>
```

Figure 6.1: Constraint in XML

A *consistency rule* consists of two parts: the first part is a header that contains a `description` element and various metadata annotations. No metadata is shown in the figure, but the language basically permits the use of arbitrary metadata tags, for example author information or comments. The second, starting in the example with a `forall` element contains the actual constraint. Rules can contain one further optional element that enables the user to control link generation behaviour. By default, both consistent and inconsistent elements are linked, but this behaviour can be overridden using a `linkgeneration` element.

```
<xlinkit:LinkBase
    xmlns:xlinkit="http://www.xlinkit.com"
    docSet="file://DocumentSet.xml"
    ruleSet="file://RuleSet.xml">
    <xlinkit:ConsistencyLink
        ruleid="rule.xml#/id('r1')">
        <xlinkit:State>consistent</xlinkit:State>
        <xlinkit:Locator
            xlink:href="advert1.xml#/Advert[1]"/>
        <xlinkit:Locator
            xlink:href="catalogue.xml#/Catalogue/Product[1]"/>
    </xlinkit:ConsistencyLink>
</xlinkit:LinkBase>
```

Figure 6.2: Sample linkbase in XML

The consistency links that are generated as a result of a check are also presented in XML, in the form of XLink *linkbases.* Figure 6.2 shows a sample linkbase containing only one XLink, which was produced as the result of checking the constraint in Figure 6.1 The link indicates that it is connecting two *consistent* elements. It contains two locators that reference the elements using a URL and an XPath expression. The remaining attributes required in an XLink have been omitted here as they are defined by default in the linkbase DTD.

6.2 Document Management

The selection of documents and rules to be checked against each other has to be managed. It is not feasible to always check every document against every rule and it is certainly not necessary to check every document every time. In the running example, people responsible for marketing may be interested in the status of adverts, whereas a customer relations department may be interested in the status of customer reports. Some support for partitioning documents and rules is needed to support flexible consistency management.

A *document set* references documents that are loaded from underlying data sources and a *rule set* references rule files that contain several consistency rules. A document set can then be submitted to be checked against a rule set.

```
<DocumentSet>
        <Document href="catalogue.xml"/>

        <Set href="Adverts.xml"/>
        <Set href="Customers.xml"/>
        <Set href="Services.xml"/>
</DocumentSet>
```

Figure 6.3: Sample document set

Fig. 6.3 shows a sample document set. Document sets form a hierarchy of documents and possibly further document sets. In the figure, the `Document` element is used to add the catalogue file directly into the set while the `Set` element includes further sets. At check time, the hierarchy is flattened and resolved into a single set. To find out whether a document needs to be checked against a rule, we check if the XPath expressions in the rule can be applied.

In order to abstract from the underlying storage mechanisms and to include data that is not natively stored in XML into a check, we provide *fetchers*. A fetcher liaises with some data store and provides a DOM tree representation of its content. By default, data are retrieved from XML files either from the disk or through an HTTP URL request using the `FileFetcher`, however user-defined fetchers can override this behaviour.

As an example, we have implemented a JDBC fetcher, which executes a query on a database and translates the resulting table into a DOM tree. Figure 6.4 shows a version of Wilbur's document set where the service reports have been put into a relational database. The `fetcher` attribute in the `Document` element overrides the default behaviour to select the `JDBCFetcher`. Several additional parameters have to be specified. The `reference` element gives a name for the root element of the output XML document, the `query` contains an SQL query that selects table data from the database, and the `key` element contains a comma-separated list of column names that make up a key for the table data.

```
<DocumentSet>
    <Document href="catalogue.xml"/>
    <Set href="Adverts.xml"/>
    <Set href="Customers.xml"/>

    <Document fetcher="JDBCFetcher"
        href="jdbc:mysql://www.xlinkit.com/testdb?user=wilbur">
        <fetcher:reference>report</fetcher:reference>
        <fetcher:query>select * from report</fetcher:query>
        <fetcher:key>productcode,description</fetcher:key>
    </Document>
</DocumentSet>
```

Figure 6.4: Document set with JDBC fetcher

```
+-------------------------+-------------+--------------------------+
| productname             | productcode | description              |
+-------------------------+-------------+--------------------------+
| HARO SHREDDER           | B001        | Found a problem in ...   |
| HARO TR2.1              | B002        | Found a problem while... |
+-------------------------+-------------+--------------------------+
<report>
     <row>
          <productname>HARO SHREDDER</productname>
          <productcode>B001</productcode>
          <description>Found a problem in ...</description>
     </row>
     <row>
          <productname>HARO TR2.1</productname>
          <productcode>B002</productcode>
          <description>Found a problem while...</description>
     </row>
</report>
```

Figure 6.5: Relational table XML representation

The JDBC fetcher class executes the SQL query on the relational database and transforms the resulting table into a DOM tree. Fig. 6.5 shows a sample table of service reports fetched from Wilbur's database by executing the JDBC query from the document set. Shown below the table is the XML representation, containing one `row` element for every row stored in the table and using the column name data from the data dictionary for the element names inside the rows.

The key information that has to be provided to the fetcher demonstrates that, in general, any fetcher that accesses non-XML data has to provide its own concept of a *unique identifier*. The unique identifier can be used by the fetcher to create hyperlink locators that do not contain XPath expressions but instead use the native referencing mechanism. In the case of the JDBC fetcher the link locators consist of values for the key fields, thus uniquely identifying a row in the table data.

```
<RuleSet>
    <header>
        <description>
            All rules for Wilbur's bike shop
        </description>
    </header>
    <RuleFile href="bike_rule.xml"
       xpath="/consistencyruleset/consistencyrule"/>
</RuleSet>
```

Figure 6.6: Sample rule set

Rule sets are managed in a similar fashion. A rule set contains references to rules and further rule sets. Fig. 6.6 shows a sample rule set. A `RuleFile` element is used to specify a rule file to load and an `xpath` attribute specifies which rules from that file to actually include. The path `/consistencyruleset/consistencyrule` will match all `consistencyrule` elements included in the rule file. If that is not desired, a more specific path such as `/consistencyruleset/consistencyrule[@id='r1']` could be used, which only loads the rule whose `id` attribute is equal to $r1$.

The document and rule set mechanisms haven proven themselves to be very valuable in practice. By using a referencing scheme and not including the rules themselves directly into a check, we enable a "pick-and-choose" approach to checking. This affords the user considerable flexibility and promotes constraint reuse. For example, it becomes possible to vary the strictness of a check by including subsets from a set of rules, or to vary the rules to be used depending on the state of a workflow engine.

6.3 Architecture and Deployment

In the abstract, our proposed checking architecture is very simple, Figure 6.7 shows its basic operation. Document sets and rule sets are submitted for checking, documents are loaded as DOM trees using fetchers, and linkbases are produced as output.

Due to its simplicity, this abstract architecture can be straightforwardly deployed in a number of different ways. Figure 6.8 shows the architecture of a public, freely accessible service that we have provided for people to check their own documents over the Internet. The check engine is implemented as a Java Servlet [Coward, 2001] and executes on the Apache Tomcat servlet engine. Users are presented with a web form where they enter the document set and rule set URL and submit the check request.

When the form is submitted, a new servlet instance is created to deal with the request. The servlet uses the Xerces XML parser from the Apache XML project to parse the documents and rule files. After checking the rules, the servlet writes an XML file containing the

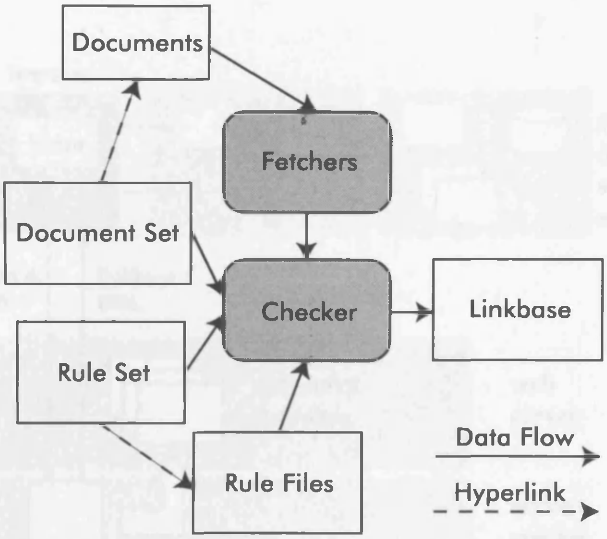


Figure 6.7: Checking component abstract architecture

generated links to the web server's local storage. It then generates a result page that contains the URL of the link base and returns it back to the browser client. The input form also gives the user a choice of whether to return the raw XML file containing the links or to add a processing directive for it to be translated into HTML using a style sheet. We will see this and further options for visualisation in Section 6.4.

Another deployment model we have implemented was to hook our checker into a messaging infrastructure. Messaging middleware has become very popular in large organisations that have extensive back-office processing and integration environments. Messaging standards such as the Java Message Service (JMS) [Hapner et al., 1999] provide asynchronous and persistent message delivery. Software engineers can make use of the middleware primitives to construct loosely connected architectures that resemble processing pipelines, while at the same time exploiting the strong delivery guarantees of the middleware.

Figure 6.9 shows how the checking component plugs into a JMS system. The checker is implemented as a Message-Driven Enterprise JavaBean [Matena and Hapner, 1999]. Whenever a set of documents needs to be checked, it is put on the checker input queue. The container then sends it on to the bean as a JMS text message. The bean performs the check and hands the resulting linkbase to a *Classifier.* Depending on the status of the links in the linkbase, the Classifier can decide which message queue to route the result to. It can also ask a *Formatter* to provide a suitable output format for the target message queue. For example, we may want to route the entire set of documents on for further processing if there is no problem, or route the linkbase if we do find an inconsistency.

This messaging deployment is interesting because it opens up the way to using consistency checking as an intelligent *routing mechanism.* Advanced architectures could be built that

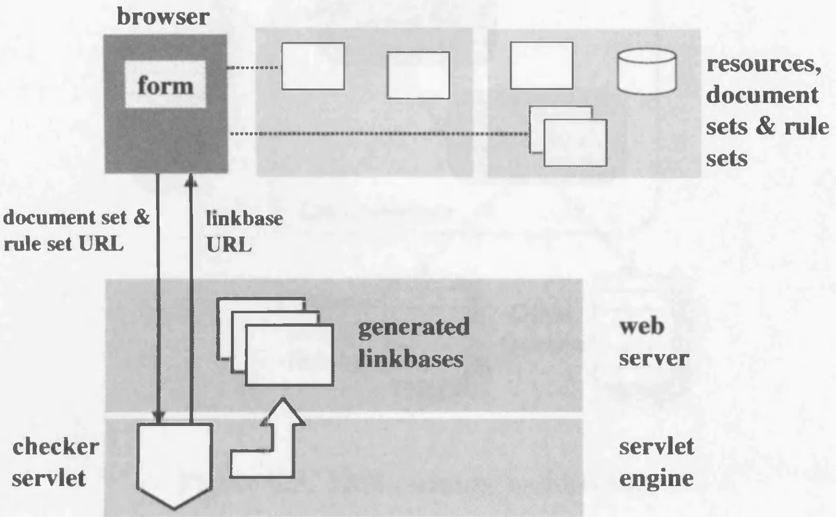


Figure 6.8: Web checking service architecture

route documents to different destinations depending on whether they fail rules, or even if they fail any particular rules. Initial conversations with stakeholders in industry suggests that this may indeed be a fruitful area for additional work, and could help industries that strive for increased automation while having to deal with data quality issues.

6.4 Advanced Diagnosis

The hyperlinks generated by the check engine are of high diagnostic value because they precisely connect the elements involved in an inconsistency. For practical purposes linkbases are of more limited value as they do not present the results in terms of the checked data. We have investigated some options for post-processing the result produced by the check engine in order to improve feedback.

6.4.1 Using the Linkbase Directly

The first option is to render the linkbase into a more interactive HTML format. Figure 6.10 shows a screenshot of a prototype implementation of such a system. The linkbase is rendered into three browser frames by a Java Servlet. The top frame displays the hyperlinks. The user can then select a pair of locators in the link, and the servlet juxtaposes the referenced elements at the top of the bottom two frames. This permits the user to compare the values of the elements in order to determine the cause of the inconsistency.

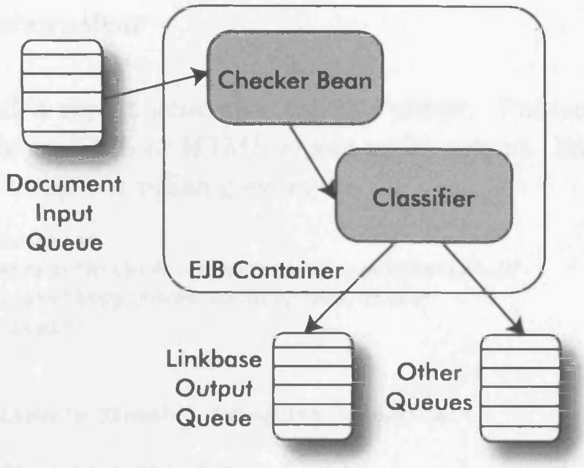
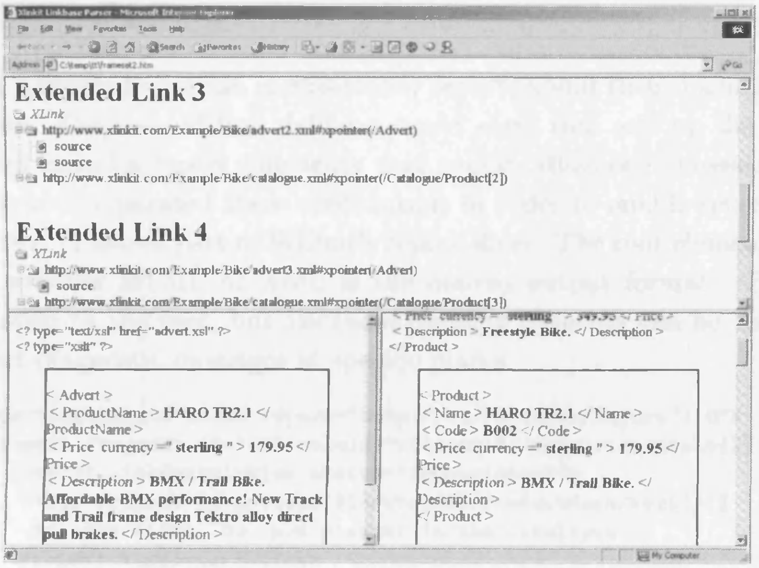


Figure 6.9: JMS message architecture

The value of this dynamic linkbase display is primarily to users familiar with the XML encoding of their data, and to rule writers who can quickly inspect the effects of executing a rule.


Xlinkit Linkbase Parser - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\temp\tt\frameset2.htm
Go
My Computer

Extended Link 3

XLink
http://www.xlinkit.com/Example/Bike/advert2.xml#xpointer(/Advert)
source
source
http://www.xlinkit.com/Example/Bike/catalogue.xml#xpointer(/Catalogue/Product[2])

Extended Link 4

XLink
http://www.xlinkit.com/Example/Bike/advert3.xml#xpointer(/Advert)
source
http://www.xlinkit.com/Example/Bike/catalogue.xml#xpointer(/Catalogue/Product[3])

```
<? type="text/xsl" href="advert.xsl" ?>
<? type="xslt" ?>

< Advert >
< ProductName > HARO TR2.1 </
ProductName >
< Price currency =" sterling " > 179.95 </
Price >
< Description > BMX / Trail Bike.
Affordable BMX performance! New Track
and Trail frame design Tektro alloy direct
pull brakes. </ Description >
```

```
< Price currency =" sterling " > 349.95 </ Price >
< Description > Freestyle Bike. </ Description >
</ Product >

< Product >
< Name > HARO TR2.1 </ Name >
< Code > B002 </ Code >
< Price currency =" sterling " > 179.95 </
Price >
< Description > BMX / Trail Bike. </
Description >
</ Product >
```

Figure 6.10: Dynamic linkbase view

The hyperlinks in a linkbase can also be used directly. By reinserting the links into the files that they are referencing, we can use the check engine results for automatic link generation. The hyperlinked files can then be styled and rendered for web publication. The case study in Section 7.1 shows how this can work in practice.

6.4.2 Report Generation

We have implemented a report generator called *Pulitzer*. Pulitzer takes as its input a linkbase and produces an XML or HTML report as its output. We will demonstrate the operation of Pulitzer using our running example.

```
<report:reportsheet
    xmlns:report="http://../Pulitzer/ReportSheet/1.0"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    output="html">
  <html>
    <body>
      <h1>Wilbur's Bikeshop Integrity Report</h1>
      <hr/>
      <h2>Advert problems</h2>
      <ul>
        <report:includefragment xlink:href="report_inc.xml#id('r2')"/>
      </ul>
      ..
    </body>
</html>
</report:reportsheet>
```

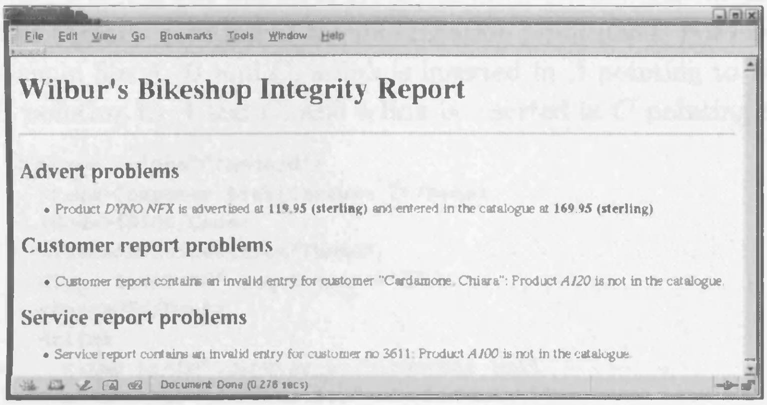
Figure 6.11: Wilbur's Pulitzer report sheet

Suppose Wilbur's decide to publish inconsistency reports about their documents as HTML pages on the web. They would first define a *report sheet* that sets up the overall layout of the report and includes *report fragments* that contain diagnostic messages for specific constraints. We have separated these mechanisms in order to enable reuse of diagnostic messages. Figure 6.11 shows part of Wilbur's report sheet. The root element of the report sheet specifies whether HTML or XML is the desired output format. The content of the element is open to the user, but `includefragment` elements can be used to instruct Pulitzer to insert diagnostic messages at specific places.

```
<report:fragments xmlns:report="http://../Pulitzer/Report/1.0">
  <report:fragment id="r2" ruleid="rule.xml#//consistencyrule[2]">
    <report:linkdescription status="inconsistent">
      <li/>Product %@=gettext($locator[1]/ProductName/text())%
      is advertised, but not present in the catalogue.
    </report:linkdescription>
  </report:fragment>
  ...
</report:fragments>
```

Figure 6.12: Report fragments for the running example

Figure 6.12 shows the report fragment describing the sample rule used in Section 5.2. The fragment specifies which rule it is describing and contains a `linkdescription` element that is applied to every inconsistent link. This element may contain arbitrary markup, as well as Pulitzer commands escaped by the `%@=` escape sequence. In the figure, the `gettext` command is used to obtain data relative to the first link locator in the linkbase from the referenced advert document.


File Edit View Go Bookmarks Tools Window Help
Document Done (0.278 secs)

Wilbur's Bikeshop Integrity Report

Advert problems

- Product *DYNO NFX* is advertised at **119.95 (sterling)** and entered in the catalogue at **169.95 (sterling)**

Customer report problems

- Customer report contains an invalid entry for customer "Cardamone, Chiara": Product *A120* is not in the catalogue.

Service report problems

- Service report contains an invalid entry for customer no 3611: Product *A100* is not in the catalogue.

Figure 6.13: Generated report in a web browser

The report sheet can now be applied to a linkbase. Figure 6.13 shows a screenshot of the generated report in a browser window.

Pulitzer is a generic tool for turning linkbases into other XML representations. As such, it is useful for presenting the results of a check in a format that other applications can understand. This is important, as documents are frequently manipulated using tools rather than directly in XML editors. By transforming the linkbases into a format that these tools understand, we can render the checking process and XML encoding transparent.

6.4.3 Link Folding

It is possible, using a linkbase processor, to take the "out-of-line" links that the check engine generates and to "fold" them into the documents that they are pointing to.

```
<xlinkit:ConsistencyLink ruleid="curr_rule.xml#//consistencyrule[1]">
  <xlinkit:State>consistent</xlinkit:State>
  <xlinkit:Locator xlink:href="curricula.xml#/.../Course[1]"/>
  <xlinkit:Locator xlink:href="computer_architecture.xml#/syllabus/identity[1]"/>
</xlinkit:ConsistencyLink>
```

Figure 6.14: Out of line link in linkbase

The linkbase processor makes copies of all documents referenced in the linkbase and inserts hyperlinks into these copies. For each hyperlink, it visits all locators and inserts back-references at the element that the locator points to. For example, the linkbase link in Figure 6.14 points to elements in `curricula.xml` and `computer_architecture.xml`. Processing this link causes a new link to be inserted in a copy of `curricula.xml`, pointing to the `identity` element in the other file, as shown in Figure 6.15. Similarly, a link is inserted in `computer_architecture.xml`, providing the reverse reference.

In the general case, where a link has an arbitrary number of locators, the processor inserts all locators except the one pointing to the modification point itself. For example, if a link points to locations in file A, B and C, a link is inserted in A pointing to B and C, a link is inserted in B pointing to A and C, and a link is inserted in C pointing to A and B.

```
<Course value="Standard">
  <Name>Computer Architecture I</Name>
  <Code>1B10</Code>
  <Theme>Architecture</Theme>
  <Type level="F" requirement="C"/>
  <Dept>CS</Dept>
  <xlink
    xlink:href="computer_architecture.xml"
    xlink:role="destination" xlink:title=""
    xlink:type="simple"
    xmlns:xlink="http://www.w3.org/1999/xlink/namespace/"/>
</Course>
```

Figure 6.15: Link inserted into document using linkbase processor

This approach can be used in a number of ways: it enables the use of our checker as an automatic link generator. Documents can be linked by checking constraints, then processed, and finally style sheets can be applied to create automatically hyperlinked web portals. Since the links are generated in response to constraints, they are guaranteed to be correct. We will see in Section 7.1 that this approach has several advantages over manual link authoring.

We can make use of inconsistent links in a similar fashion. Instead of providing links that are guaranteed to be correct, we can establish a "web of inconsistency". We can apply style sheets or augment third party applications with proprietary processing so as to allow users to navigate between inconsistent parts of documents.

6.4.4 Tight Tool Integration

Another option is to exploit the plug-in and scripting mechanisms present in many modern document editing tools in order to display inconsistency reports. If the data import facilities of a particular tool are not rich enough to process hyperlinked input files, this may be the only available option.

The obvious advantage of this approach is that it offers a high degree of transparency. Consistency links and diagnostic messages can be interpreted and displayed in a fashion that is suitable for the application, including reacting to user events typical for the application. As an example, a UML editor could provide a popup over each model element that takes the user to elements that are participating in an inconsistent relationship.

The downside of tight integration is the relatively high overhead in implementing such a system. In addition the implementation is proprietary to each tool, and a separate implementation is needed for each tool.

6.5 Chapter Summary

This chapter presents a few of the implementation details of our check engine. We have given an example of a constraint in XML form. We have also looked at the rule and document management mechanisms that we have provided: document sets, which group documents as a unit for check submission, and rule sets, which group rules to apply to documents. We have introduced *fetchers* as simple programs that transform non-XML input to XML in the course of retrieving data for checking.

We have shown that architecturally, our check engine can be deployed in a number of guises, whether it be a command line application that checks a document set against a rule set, a servlet that is provided with documents over the Internet, or a message processing component that retrieves documents for checking and routes them on depending on the result. Adapting the checking component for different architectures is rather straightforward, as it is a well-encapsulated component with a clearly defined interface – to take some documents and constraints, and produce a check result.

Finally, we have investigated some ways of transforming the output of the check engine from its raw linkbase format for further presentation or processing: visualisation components can help the user to navigate between the linked portions or documents, reporting can use the links to provide diagnostic messages, linkbases can be "folded" back into documents to provide automatic hyperlinking, and they can also be used for tool integration, to provide diagnostic messages in tools that are used for document manipulation.

This chapter has thus afforded us a look at what has gone into a practical implementation of our linking semantics inside our check engine, and, most importantly, has demonstrated that both architecturally, and in terms of its processing output, it is flexible and can be readily adapted for diverse applications. None of the implementations we have completed has been overly difficult, mostly due to the use of open and standardised technology. This gives us hope that our design decisions, coupled with the notion of consistency checking as a service, will not be overly complicated for practitioners to adopt and will integrate without too much effort into their own systems.

7 Case Studies

We have seen how to derive a novel form of diagnostics, in the form of consistency links, and how these diagnostics can be put to use in the implementation of a simple consistency checking service. The critical question for judging the validity of this approach must be whether it can cope with the heterogeneity, distribution and diagnostic requirements of a real-life setting.

We have applied our checker in a number of case studies, in such diverse areas as finance, software engineering, bio-informatics and university curriculum information. By choosing a wide range of domains we reassure ourselves that our approach can indeed handle varying requirements and is flexible and adaptable. In this chapter we will look at two comprehensive examples, the checking of a university curriculum and a detailed investigation of the problem of managing heterogeneous software engineering artifacts in the area of Enterprise JavaBeans development.

7.1 Case Study: Managing a University Curriculum

The Department of Computer Science at University College London introduced a new curriculum and associated course syllabi. In order to provide high quality information in the wide variety of different representations required, it decided to adopt XML as a common format. The system has to hold a curriculum and provide links to the syllabi for students, depending on which degree programme they are pursuing. Figure 7.1 shows a sample abbreviated syllabus file for a course. Each course syllabus is held in a separate XML file. The overall curricula for degree programmes are all kept combined in one file. For each degree programme, the mandatory and optional courses are listed, grouped by the year in which they can be selected.

The process of syllabus development is highly decentralised, with different people providing additions and corrections to course syllabi. Curriculum files contain information related to the individual syllabus files. For example, course codes mentioned in the curriculum files have to be part of a syllabus definition. The study also called for a "study plan" feature, simple XML files in which students could list the courses they were taking. These study plans also had to be consistent with the curriculum, referencing existing courses and including the correct mix of optional and mandatory courses. Altogether, ten rules were initially identified as necessary to preserve the consistency of the system. The complete set of English language rules can be found in Appendix C.

It is desirable for navigation purposes to provide hyper-links from the curriculum to individual courses. However, manually adding links from the curriculum file to all 48 syllabus

```
<syllabus>
    <identity>
        <title>Concurrency</title>
        <code>3C03</code>
        <summary>The principles of concurrency
                  control and specification</summary>
    </identity>
    <subject>
        <prerequisites descr="">
            <pre_code>1B11</pre_code>
        </prerequisites>
    </subject>
</syllabus>
```

Figure 7.1: Sample shortened syllabus file in XML

files would be error prone as files get deleted and courses renamed. It is preferable to use the semantically equivalent information in the files (e.g. the course codes) to generate the hyperlinks automatically. We used the check engine to achieve both goals.

Figure 7.2 shows the time used for checking each rule against all 52 documents. The syllabus files were all around 5 kilobytes in size and the curriculum is 110 kilobytes in size. Checking was performed on a 700 Mhz Intel Pentium 3 machine with 128Mb of RAM, running Mandrake Linux 8.0 with kernel 2.4.9 and the IBM JDK 1.3. The total checking time was 11.1 seconds, with the most complex rule taking 8 seconds to check. In total, 410 consistent and 11 inconsistent links were generated.

The exceptional checking time on rule 5 was caused by a transitive closure operation. Transitive closure is not a part of the XPath function library, so we had to implement a simple extension function. The implementation was only for prototyping purposes and uses a rather naive algorithm, leading to the long check time.

The second goal of the case study was to provide a fully linked HTML version of the department's curriculum to be browsed by staff and students. One of the rules for the curriculum is that every course listed in the curriculum must have a syllabus definition. If the rule is satisfied, a consistent link is generated from the course entry in the curriculum to the syllabus defining the course. We used XTooX, an XLink linkbase processor, to fold all consistent links from this rule back into the XML file containing the curriculum. We then only had to provide a simple XSL stylesheet that transforms the XML file and simple links into an HTML representation. Figure 7.3 shows the "production version" of the curriculum website.

The final goal was to highlight inconsistencies in the relationships between the syllabus and curriculum files. Due to the large number of files, mistakes are easily possible and the check during the case study resulted in five real errors being highlighted. Figure 7.4 shows a screenshot of the inconsistency report that was generated by applying a report

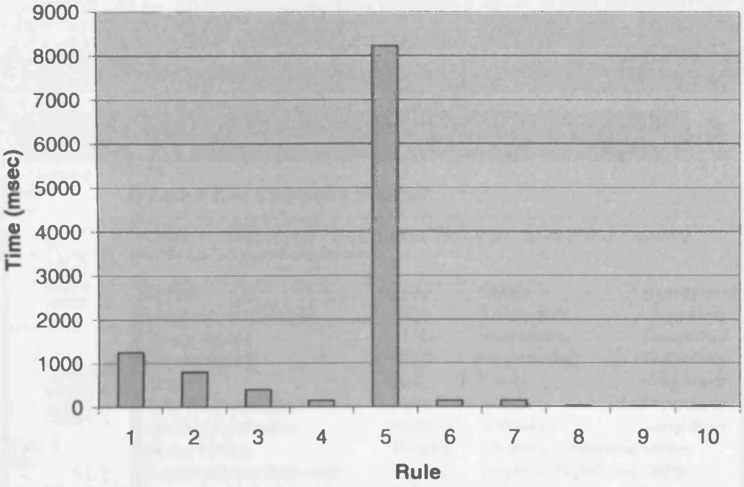


Figure 7.2: Syllabus study timings

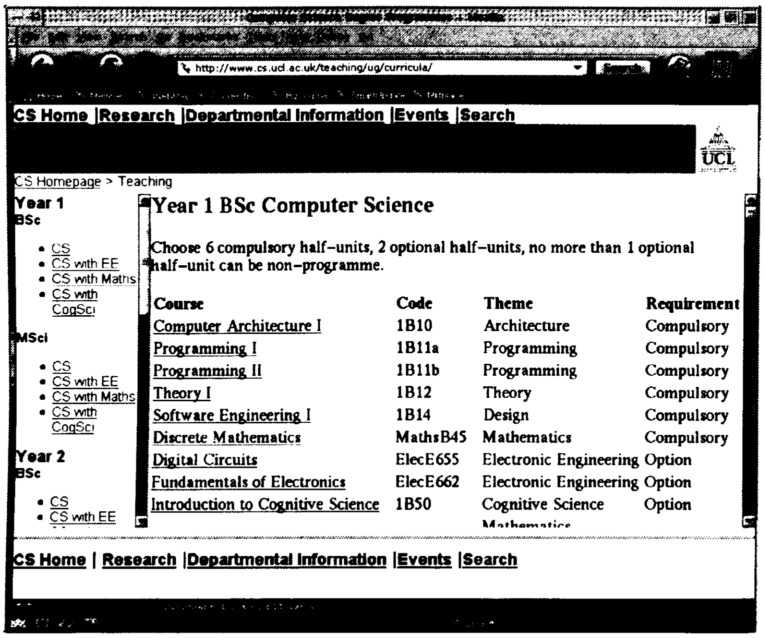
sheet to the linkbase generated by the check – see Section 6.4 for details on report sheets. The report messages highlight those courses that reference prerequisite courses that were not defined as part of the curriculum.

This case study is unusual in that it used the check engine both as a link generator and for its original purpose as a consistency checker. By generating hyperlinks between the 52 files in an automated fashion, the error prone and tedious manual linking activity could be eliminated, and a simple web portal constructed and regenerated quickly. The inconsistency reports help with the management of the plethora of files. They are convenient to edit individually, but the sheer volume of files meant that it had previously been hard to maintain an overview of the consistency of the collection as a whole. We were thus able, using our architecture, to avoid the need to integrate into a repository and were able to support the existing infrastructure and working practices.

7.2 Case Study: Software Engineering

Our second case study looks at the problem of checking the consistency of software engineering documents. Software engineering provides a challenging benchmark as the documents and the constraints they have to obey tend to be quite complex in both structure and size. We have chosen for the study the problem of checking the various artefacts that are produced during Enterprise JavaBeans [DeMichiel et al., 2001] development.

In order to bring structure to the discussion, we first attempt a categorisation of the consistency constraints that arise during software development. From these we derive a set of requirements for consistency management services in this particular application



http://www.cs.ucl.ac.uk/teaching/ug/curricula/

CS Home |Research |Departmental Information |Events |Search

CS Homepage > Teaching

Year 1
BSc

- CS
- CS with EE
- CS with Maths
- CS with CogSci

MSci

- CS
- CS with EE
- CS with Maths
- CS with CogSci

Year 2
BSc

- CS
- CS with EE

Year 1 BSc Computer Science

Choose 6 compulsory half–units, 2 optional half–units, no more than 1 optional half–unit can be non–programme.

Course	Code	Theme	Requirement
Computer Architecture I	1B10	Architecture	Compulsory
Programming I	1B11a	Programming	Compulsory
Programming II	1B11b	Programming	Compulsory
Theory I	1B12	Theory	Compulsory
Software Engineering I	1B14	Design	Compulsory
Discrete Mathematics	MathsB45	Mathematics	Compulsory
Digital Circuits	ElecE655	Electronic Engineering	Option
Fundamentals of Electronics	ElecE662	Electronic Engineering	Option
Introduction to Cognitive Science	1B50	Cognitive Science	Option
		Mathematics	

CS Home | Research |Departmental Information |Events |Search

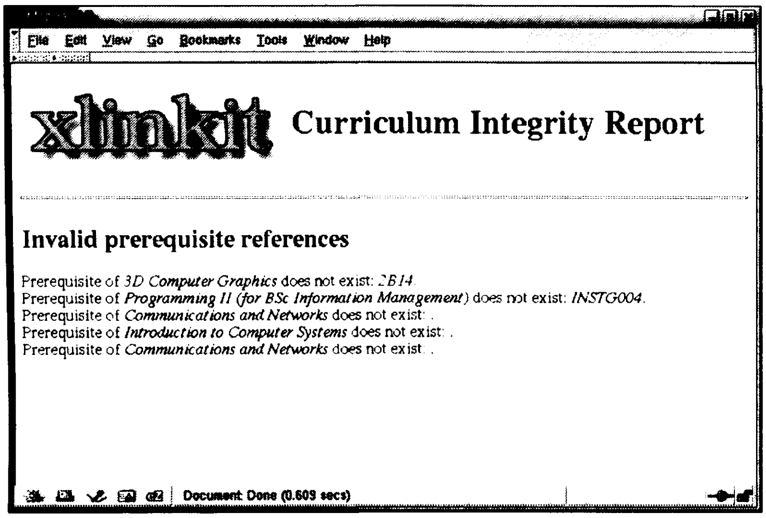
Figure 7.3: Automatically generated links in the curriculum

scenario. The case study then demonstrates how our checker can be used to address these requirements during the development of systems based on Enterprise JavaBeans. We show checks between design, implementation and deployment information and provide an evaluation of the results.

7.2.1 Constraints in Software Engineering

Throughout a software development lifecycle, many types of consistency constraints appear, including *horizontal* constraints between artefacts at the same stage of the lifecycle and *vertical* constraints between stages, for example between the design and implementation. Developers are aware of such constraints, yet they are rarely explicitly expressed let alone automatically checked. Instead, ad-hoc techniques such as inspection and manual comparisons are used to maintain consistency. It is our goal to aid developers in the task of managing consistency, by providing them with a more systematic approach and tools for automation. In order to achieve this, we first categorise the types of constraints that occur in practice, and then derive a set of requirements for tools that provide support for these types of constraints.

In our categorisation, we use the Unified Modeling Language (UML) [OMG, 2000a] as a guiding example. The UML is widely used in software development. It combines a graphical notation with a semi-formal language, based on the OMG's Meta-Object Facility (MOF) [Object Management Group, 2000a]. The language and notation provide support



Curriculum Integrity Report

Invalid prerequisite references

Prerequisite of *3D Computer Graphics* does not exist: *2B14*.
Prerequisite of *Programming II (for BSc Information Management)* does not exist: *INSTG004*.
Prerequisite of *Communications and Networks* does not exist: .
Prerequisite of *Introduction to Computer Systems* does not exist: .
Prerequisite of *Communications and Networks* does not exist: .

Document: Done (0.603 secs)

Figure 7.4: Curriculum inconsistency report screen shot

for capturing a system from a variety of perspectives, including static perspectives such as class diagrams and dynamic ones such as sequence diagrams.

Specification languages used in software engineering typically come with a set of static constraints that valid specifications have to obey. These constrain the relationships between the elements of the underlying semantic models of the language. We call such constraints *standard constraints*, because they are generally standardised when a specification language is formally defined. For example, the UML has a syntax, and models expressed in it have to obey static semantic constraints. The UML specification expresses these constraints in the Object Constraint Language (OCL) [OMG, 2000a]. If any of the constraints are violated, by definition inconsistency is introduced into the model.

When putting any specification language, including the UML, to practical use it very frequently transpires that the expressive capabilities provided by the language are not sufficient for a particular purpose and require extension or adaptation. The specification language may not be semantically rich enough to capture all the information necessary, for example many design languages obscure architectural information or design patterns. Conversely, the language may be overly expressive and insufficiently concise for a particular practical purpose, for example many languages lead to specifications that are too complex to be shown to stakeholders. These problems can sometimes be circumvented by restricting the language or introducing naming conventions for specification artefacts. Such restrictions or conventions will invariably give rise to additional semantics not catered for by the language. It is thus important to properly specify the extension method, and to ensure that it is applied consistently. This can be accomplished by introducing *extension constraints* that supplement the standard constraints of a language.

In the case of the UML, it was recognised that the basic metamodel of the UML was not rich enough to express the architectural properties of distributed systems that build on middleware. The UML standard foresees such problems and includes a general extension framework centered around *stereotypes* and *tagged values*, which can be used to annotate model elements in a UML model with additional information. Certain extensions are peculiar to a domain, such as the domain of systems built on middleware, and have been standardised: the UML Profile for EJB [Matena and Hapner, 1999] lists a set of stereotypes and tagged values for designing systems built on top of Enterprise JavaBeans. In a similar fashion, the OMG's UML Profile for CORBA [Object Management Group, 2000b] provides features for CORBA-based systems [Object Management Group, 2001]. These extension frameworks also supply a set of constraints that specify the properties that model elements annotated with the standardised stereotypes have to obey.

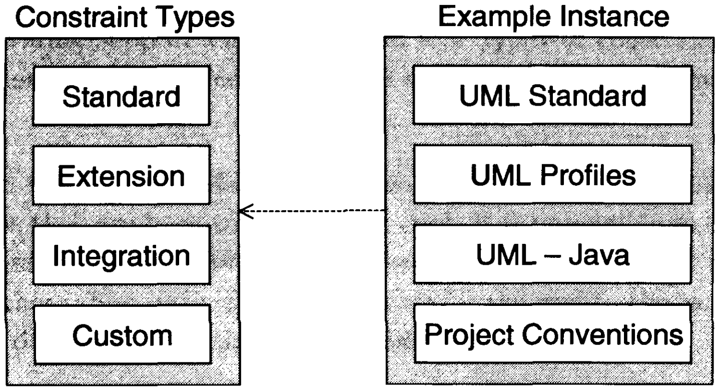


Figure 7.5: Constraints types with examples

Two further classes of constraints often appear at different stages of the development lifecycle. For example, if we wish to check that the implementation of a system is consistent with the design, we have to introduce additional constraints that relate them. Or if we wish to refine the classes in our UML class diagrams into Z schemas, we need constraints between the Z specification and the UML model. In both examples, constraints arise naturally as the different specification or implementation languages denote the same elements of the semantic model. We refer to these constraints as *integration constraints*. Integration constraints also arise when specification languages are restricted to interoperate cleanly with one another: when using the UML to design a Java-based system, we may want to check that the design does not include any multiple inheritance between classes.

Finally, developers frequently introduce additional constraints for their specifications. For example, software development organisations may introduce conventions that they want their software engineers to adhere to, the most prominent example being naming conventions. These *custom constraints* may be defined to ensure consistency between different products produced by the company, or be specific to a certain project. Such constraints are mostly informally specified and checked manually. Figure 7.5 gives an overview of the types of constraints that we have identified, and gives a UML-based project as an example instance.

7.2.2 Tool Support

Effective use of specification languages in the construction of large and complex systems requires CASE tools, which often include mechanisms for checking the constraints of the language they are built to support. A good CASE tool will let its user check on demand whether a specification is consistent with respect to the constraints or will provide interactive checking as changes are made. The majority of CASE tools currently on the market provide rather static support for consistency checking, and most support only standard constraints. Providing support for checking all four types of constraints, standard, extension, integration and custom constraints, is not trivial and gives rise to a number of problems.

The first problem is flexibility. Many CASE tools still handle only standard constraints, and in many cases they are hard-coded into the tool. It is impossible to support the three other types of constraints with such tools. These tools generally treat all constraints as equal and apply them all at once. It is not possible for the user to choose to ignore certain constraints or to delay the resolution of inconsistency.

To take an example from the UML, the constraint that no inheritance cycles must be present in a class diagram expresses a concept fundamental to object-oriented technology and must hold in all models. By contrast, the constraint that requires all attributes in classes to have valid types is often consciously violated. The UML is used for a variety of purposes, ranging from high-level domain analysis models to detailed designs. It does not make sense to apply the same set of constraints to all models – some flexibility in choosing constraints is required. While there are commercial tools that offer options for delaying the resolution of problems – Argo/UML's [Robins et al., 1999] critic systems, which is based on earlier resarch on cognitive support in software development is one example,– many UML tools still take a static approach to inconsistency handling. This lack of flexibility represents a barrier if all four types of constraints are to be supported.

In addition, there is still a large number of CASE tools that *enforce* their constraints, further undermining the flexibility needed in software development. This behaviour imposes a process on the developer, who has to add objects to the specification in a certain order so as to maintain consistency at all times. It would be preferable to provide a checking service that notifies developers of inconsistency, but gives them the option not to address it. This tolerant approach to inconsistency becomes more important as development artefacts at different stages of the lifecycle are checked against each other, as maintaining total consistency at all times between stages is both unnecessary and time consuming [Nuseibeh et al., 2000].

Secondly, checking integration constraints is a rather more difficult problem than checking any of the other types of constraints. Since specifications expressed in different languages are to be checked against each other, the problem of language heterogeneity must be addressed. It is possible to provide support for integration constraints in each individual CASE tool, by adding support for addressing objects in all other languages, but this approach is clearly unworkable. It thus makes sense to provide a checking service that does not reside in any one tool, but *between* tools. Such a checking service can bridge the heterogeneity gap between the tools by either reducing all specification languages to a common vocabulary such as first order logic – a difficult process that may cause prohibitive overhead – or by providing a mechanism for expressing relationships between artefacts in different languages directly.

Thirdly, proper support for integration and custom constraints requires the problem of distribution to be addressed. Developers working on a large and complex system often do not store their specifications, implementation and deployment artefacts in one central repository. In order to fully support integration constraints between heterogeneous artefacts, we thus have to assume that these artefacts may be distributed, and perform a distributed consistency check. This kind of mechanism is clearly beyond the state of the art of current tools, most of which do not even provide mechanisms for splitting up and distributing individual specifications, let alone heterogeneous artefacts.

Finally, many constraint languages have been designed to be very expressive, at the cost of making it hard to provide diagnostic feedback. For example, most tools that provide support for OCL can only supply information on whether a constraint has been violated or not. OCL constraints are attached to model elements, so a constraint can be either *true* or *false* for that particular element. This is inadequate as it imposes the burden of finding the cause of inconsistency on the developer – it does not state why, for example due to the presence of other information, the inconsistency has occurred.

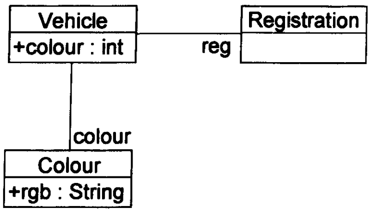


Figure 7.6: Inconsistent UML model

As an example, the UML includes a constraint that prescribes that no attribute name in a class must be equal to a role name on the opposite end of an association connected to the class – constraint 3 for Classfiers in Section D.1. If this constraint is violated, as in Figure 7.6, and the only diagnostic information available is a pointer to the class, the developer is left to inspect all associations connected to the class. When these associations are shown in different diagrams, or not shown at all – UML elements do not have to be present in diagrams in order to be part of a model – or their number is large, this can delay the developer. Even worse, if multiple heterogeneous and distributed specifications are checked against each other, the overhead of browsing through all of them in search of what caused the inconsistency will be prohibitive. It is necessary in this setting to provide diagnostics that pinpoint exactly the combination of elements in all specifications that cause an inconsistency.

7.2.3 Consistency in EJB Development

We now examine in detail how the different types of constraints we have identified arise in practice and how they can be addressed. In this case study we use the check engine to check the standard, extension, integration and custom constraints that arise in different stages, and between stages, of the development life-cycle of a system based on Enterprise JavaBeans. We precede the study with a short overview of Enterprise JavaBeans.

Enterprise JavaBeans [Matena and Hapner, 1999] is a popular technology for component-based software development. Components are implemented as *beans* that reside in a *container*. The fundamental idea underlying EJB is that the container should provide all the infrastructure services for the bean, such as transaction management, persistence, replication and communication with the underlying middleware, so that the developer can concentrate on implementing business logic.

Implementing an EJB requires the developer to supply the following artefacts: A *remote interface* specifying the methods the bean supplies to client objects, a *home interface* specifying the methods the bean supplies to its container, and a *bean implementation*, containing the business logic and event methods that are called when the life cycle status of the bean changes, for example when it is created and destroyed. A distinction is made here between *entity beans*, which represent persistent data, and *session beans*, which implement control flow.

When the bean is ready to be deployed into the container, the developer must further provide a *deployment descriptor*, expressed in XML, that describes the type of the bean and its quality-of-service contract with the container – for example whether persistence of the bean's data will be managed by the bean or by the container.

In the case study, the Unified Modeling Language (UML) [OMG, 2000a] was used as the design language of choice. While the UML is semantically rich enough to express the implementation details of an EJB-based system, the expressiveness of the design model can be improved by clearly identifying the features supplied by the EJB architecture, that is by making the architecture explicit. It is possible by these means to distinguish the

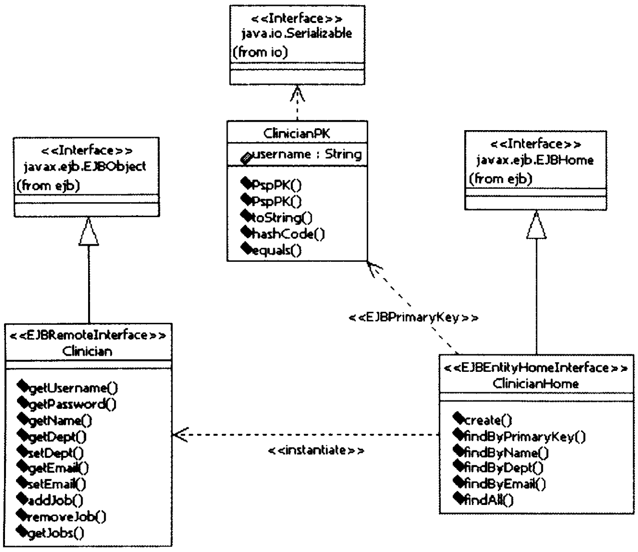


Figure 7.7: EJB-Profile compliant UML design of a single bean

high-level components supporting the EJB architecture from implementation details.

The UML Profile for EJB [Greenfield, 2001] – the "EJB profile" or just "profile" from here on – identifies a set of stereotypes that can be used to label model elements that represent EJB artefacts, and the relationships between them. Figure 7.7 shows a fragment of a UML model annotated in this way. Since the stereotypes expose new semantics for the model, they cannot be combined arbitrarily, but have to obey the constraints set out in the profile.

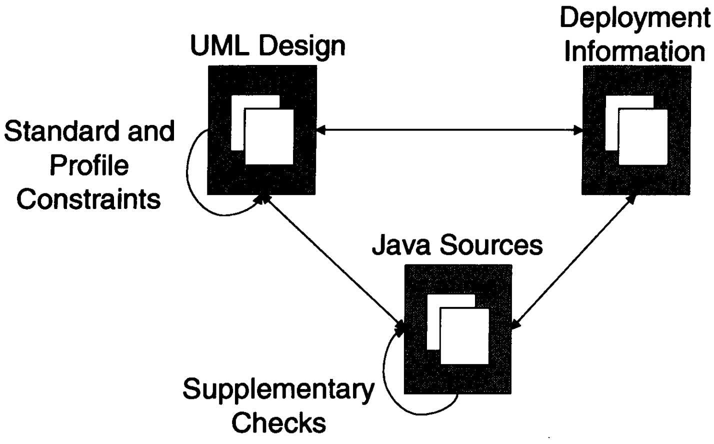


Figure 7.8: Consistency checks for EJB development

Given the UML model, the implementation and the deployment information, we can now specify a framework for managing the consistency of the system as artefacts are added, changed and removed during the lifecycle. Figure 7.8 shows the checks that we want to perform: we want to check the UML model internally, first by checking that the standard constraints set out in the UML specification are obeyed, and then that the extension constraints of the EJB profile hold; we wish to check the integration constraints between the UML model and the deployment descriptor and implementation, between the deployment descriptor and the implementation, and additional custom constraints inside the implementation. We will discuss each of these types of checks using an example taken from the study.

```
<description>
   The AssociationEnds must have a unique name within the Association
</description>
<forall var="a" in="//Foundation.Core.Association">
   <forall var="x" in="$a/Foundation.Core.Association.connection/
                       Foundation.Core.AssociationEnd">
      <forall var="y" in="$a/Foundation.Core.Association.connection/
                          Foundation.Core.AssociationEnd">
         <implies>
            <equal op1="$x/Foundation.Core.ModelElement.name/text()"
                   op2="$y/Foundation.Core.ModelElement.name/text()"/>
            <same op1="$x" op2="$y"/>
         </implies>
      </forall>
   </forall>
</forall>
```

Figure 7.9: Example of a UML *standard* constraint

Our first check will determine if the UML model is a valid model. The UML specification sets out the constraints that have to hold for each metaclass in the model. In order to enable us to constrain the models, we have to make them available in an XML format. We chose XMI 1.0 [OMG, 2000b] for this purpose as it is implemented by a number of CASE tools. We have expressed a subset of those constraints – those of the **Foundation.Core** package, which deals with static information such as class diagrams – for use in this case study. There are 34 constraints in total, they are listed in Appendix D.1. These constraints are clearly *standard constraints*. Figure 7.9 gives an example of such a constraint expressed in our constraint language.

We then check the constraints of the EJB profile against our model. We have chosen to express a subset of those constraints for this case study. This subset specifies the relationships of EJB elements that make up the "external view" of a system: remote and home interfaces, EJB methods, and "primary key classes" that provide the key attributes for entity beans. We have written most constraints in this subset, apart from the constraints on EJB methods and RMI interface inheritance. These constraints are no more complex than the ones we have expressed and would not add significant value to this study. The complete list of constraints we have expressed, 16 in total, is given in Appendix D. These constraints are *extension constraints* since they supplement or customize the UML for

```
<description>
    The (EJB entity home) class must be tagged as persistent.
</description>
<forall var="c" in="$classes">
    <implies>
        <exists var="s" in="id($c/Foundation.Core.ModelElement.stereotype/
                                Foundation.Extension_Mechanisms.Stereotype/
                                @xmi.idref)[Foundation.Core.ModelElement.name/
                                text()='EJBEntityHomeInterface']"/>
        <exists var="t" in="$c/Foundation.Core.ModelElement.taggedValue/
                              Foundation.Extension_Mechanisms.TaggedValue[
                              Foundation.Extension_Mechanisms.TaggedValue.tag/
                              text()='persistence' and
                              Foundation.Extension_Mechanisms.TaggedValue.value/
                              text()='persistent']"/>
    </implies>
</forall>
```

Figure 7.10: Example of an EJB profile *extension* constraint

use in a particular application domain. Figure 7.10 gives an example of an EJB profile constraint.

```
<description>
    Each attribute listed as a 'cmp-field' for an entity bean in the
    deployment descriptor must be an attribute of the bean implementation
    class
</description>
<forall var="c" in="$entitydescr">
    <forall var="b" in="$javaclasses[concat(../package/@name,'.',@name)=
                         $c/ejb-class/text()]">
        <forall var="f" in="$c/cmp-field">
            <exists var="v" in="$b/var[@name=$f/field-name/text()]"/>
        </forall>
    </forall>
</forall>
```

Figure 7.11: Example of a deployment descriptor – implementation *integration* constraint

We also check the UML design against the Java implementation and against the deployment descriptor, and the Java implementation directly against the deployment descriptor. These are all examples of *integration constraints* as they restrict the content of software development artefacts depending on the contents of additional artefacts. In the case of these checks, there is no standard to guide us. Indeed, we expect these checks to vary depending on the needs of developers or project managers. We have consequently only provided nine samples of the kinds of relationships that could be checked; they are also listed in Appendix D. Figure 7.11 shows a sample constraint that checks if every field in an entity bean that has been declared as a container-managed persistent field in the deployment descriptor is present as a variable in the class implementing the bean.

```
<description>
    Every remote interface is implemented by a bean class that resides
    in the same package
</description>
<forall var="i" in="/java/interface[extends/@name='EJBObject']">
    <exists var="c" in="/java/class[../package/@name=$i/../package/@name
                        and (implements/@name='EntityBean' or
                             implements/@name='SessionBean')]"/>
</forall>
```

Figure 7.12: Example of a *custom* constraint

Finally, we have specified some additional checks for the Java implementation. These checks are internal to the implementation, but do check relationships between multiple Java files. To take one example, we have specified the constraint that for each remote interface there must be a bean implementation class that implements the interface. This constraint is not checked by the Java interface implementation mechanism because EJB implementation classes do *not* implement their remote interface. This is because clients of an EJB do not directly invoke the bean itself through the interface, but instead make calls to a proxy object. It is this proxy object, which is automatically generated, that implements the remote interface. A further discussion of this pattern can be found in [Marinescu, 2002].

Figure 7.12 shows the constraint expressed in the constraint language. We have expressed two sample constraints, they are given in Appendix D. These constraints are not standardised and do not depend on the contents of further artefacts, and are thus *custom constraints* – although one could imagine that they could become extension constraints in the future.

7.2.4 Evaluation

In order to realize the checks outlined in Figure 7.8, we have to map them onto the mechanisms provided by the checker. This mapping is straightforward, Figure 7.13 shows how the documents and constraints are managed using the checker. We will first discuss the mapping process and then present the results of checking the constraints against our case study system.

We load the UML model, the deployment descriptor and the Java files into DocumentSets. The UML model can be straightforwardly loaded as an XMI file and the deployment descriptor can also be easily loaded since it is already represented in XML markup.

In order to include the Java files into the check, we have to provide an additional *fetcher* that can parse Java classes and turn them into DOM trees. This was done by using the CUP [Hudson, 1999] Java parser generator to process a standard grammar for Java whose productions had been annotated with commands to emit nodes for the tree. Fig-

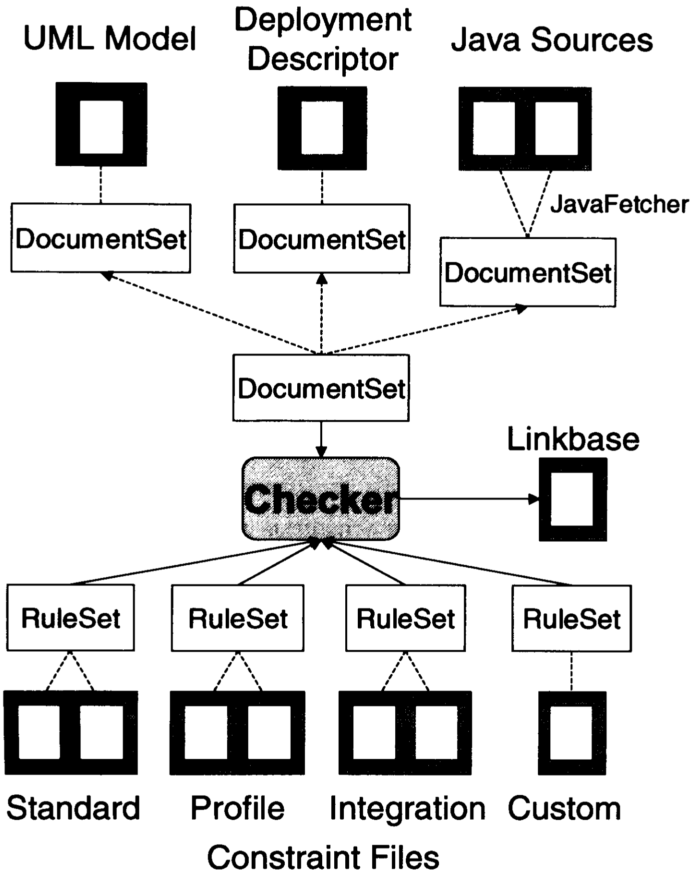


Figure 7.13: EJB consistency checks

ure 7.14 shows an XML file that was generated by parsing a small Java class, turning it into a DOM tree and serializing it to XML. We do not convert the entire Java file, including the code inside the methods, because we do not require this information in our checks. If such information was desired, recent approaches from compiler construction like XANTLR [Trancon y Widemann et al., 2001] could be used instead.

The different types of checks that we wish to perform map nicely onto RuleSets: each arc in Figure 7.8 becomes a rule set in the implementation. Each rule set references a number of rule files that contain the actual rules. The rule sets can then either be checked individually, as indicated in the figure, or assembled into one combined rule set to check all the rules.

Expressing the constraints was not straightforward at first, due to the complexity of XMI. While the logic of the constraints was straightforward, the path expressions tended to become quite verbose. Fortunately, many constraints reference the same model elements in the XMI file, making it possible to reuse most path expressions. Projects with undergraduate students have also shown that expressing constraints over XMI, while presenting

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
    <package name="uk.ac.ucl.aivetac.session.formjob"/>
    <class name="FormJobWorkFlowBean">
        <implements name="SessionBean"/>
        <method name="findHome"/>
        <method name="allProjects"/>
        <method name="allProtocols"/>
        <method name="allReports"/>
        <method name="setSessionContext"/>
        <method name="ejbActivate"/>
        <method name="ejbPassivate"/>
        <method name="ejbCreate"/>
        <method name="ejbRemove"/>
    </class>
</java>
```

Figure 7.14: Java structure representation in XML

a steep learning curve in order to understand the internal format of XMI, leads to a certain familiarity over time, allowing most students to write constraints within a couple of weeks. We encountered no problem at all expressing any of the other constraints involving the deployment descriptor or Java sources, which have straightforward XML encodings. These constraints were all expressed within a matter of minutes.

We have evaluated our constraints against a complete and working EJB-based system that was built to provide a web front-end for statistical evaluations of drug trials. The system consists of 9 EJBs, 4 session beans and 5 entity beans. The design model was annotated with the required stereotypes of the EJB profile, was 2.5 MB in size and contained 41 classes. The implementation comprised 40 Java source files. The checks were performed on an idle Intel Pentium 4 machine running at 1.4Ghz and using the IBM JDK 1.2.

We first checked the UML model separately before checking the remaining constraints. This check, against all 34 rules, and including loading and parsing the UML model, took 26 seconds averaged over three runs. The check generated 324 inconsistent links. Many of those inconsistencies were caused by associations having association ends with equal names, a constraint that is often violated in design models that show high level views, by leaving association ends unnamed. Other inconsistencies were caused by features not present in the XMI files produced by Rational Rose, such as the distinction between methods and operations. No other significant inconsistencies were found.

We then proceeded to check the extension, integration and custom constraints between the UML model, the deployment descriptor and the Java files. Checking all 26 rules against all 42 files took 23 seconds, including parsing the Java files and loading the UML models. By far the greatest amount of time was spent on checking rules that related to the UML model, due to the complexity of the XPath expressions needed to get information from XMI. None of the rules involving only Java sources or the deployment descriptor took longer than 380 milliseconds each to check.

Checking the extension, integration and custom constraints generated 94 inconsistent links. 92 of those were caused by the model not being entirely EJB-profile compliant: the methods in the home and remote interfaces did not carry any EJB method stereotypes. One other inconsistency occurred because the primary key for one entity bean was specified to be a primitive type, rather than a class type, as required by the EJB specification. And finally, a variable that had been declared in the deployment descriptor as a persistent field for an entity bean was missing in the bean implementation.

The quality of the links produced by the link generation semantics can be seen by looking at what kinds of links are produced by the sample rules we have shown in the previous subsection:

- *The AssociationEnds must have a unique name within the Association* (Figure 7.9). For associations that violated that rule, the generator produced a ternary link that connects the association and the two association ends with equal names.
- *The (EJB entity home) class must be tagged as persistent* (Figure 7.10). For classes that violated that constraint, the generator produced a link from the class to the stereotype declaring the class as an EJB entity home interface. This makes sense because in the absence of a persistence tag, the inconsistency is caused by the stereotype being present.
- *Every remote interface is implemented by a bean class that resides in the same package* (Figure 7.12). If no such bean class exists for a particular remote interface, an inconsistent link that points to the remote interface is generated.

```
<xlinkit:ConsistencyLink ruleid="javadeploy_inter.xml#//consistencyrule[@id='r2']">
    <xlinkit:State>inconsistent</xlinkit:State>
    <xlinkit:Locator
        xlink:href="/config/ejb-jar.xml#/ejb-jar/enterprise-beans[1]/entity[1]"/>
    <xlinkit:Locator
        xlink:href="/entity/protocol/ProtocolBean.java#/java/class[1]"/>
    <xlinkit:Locator
        xlink:href="/config/ejb-jar.xml#/ejb-jar/enterprise-beans[1]/
                                                entity[1]/cmp-field[3]"/>
</xlinkit:ConsistencyLink>
```

Figure 7.15: Sample inconsistency - field from deployment descriptor not implemented

As an example of what these links look like in their XLink form, Figure 7.15 shows the link the checker generated when the constraint in Figure 7.11 was violated: The `entity` declaration in the deployment descriptor is linked to the `ProtocolBean` class implementation, and the `cmp-field` (Container-Managed Persistence field) in the descriptor. The cause of the inconsistency is thus immediately obvious by looking at the rule description text and this link: the linked field in the deployment descriptor should have been added to the linked class, or removed from the descriptor.

This case study demonstrates that our checker can be used to check all four types of constraints we have identified; that it can cope with heterogeneous notations; that distribution of specifications can be addressed through a structured document management mechanism. It shows how our novel semantics for first order logic, which produces hyperlinks instead of boolean results, generates powerful diagnostics that connect inconsistent elements across specifications. And what is more, it demonstrates that inconsistencies in the design have not led to any problems in the working EJB system, which justifies our tolerant approach to inconsistency.

8 Scalability

In our research work we have tried to strike a balance between theoretical investigation and demonstration of practical feasibility. Since our background is largely in software engineering, we think that an investigation of scalability is an important part of assessing the applicability and limitations of our approach.

Scalability is a difficult problem to address in practice. Our check engine scales in a number of factors, such as document size or number of rules, and it is unlikely that we can come up with a solution that works well no matter what size of problem is presented to it. Instead we want to investigate approaches that can be used in particular scenarios and determine the effort involved in their realisation.

Previous experience shows us that addressing scalability frequently comes at a price, be it implementation complexity, architectural overhead or management overhead for the user. The purpose of this chapter is to quantify these costs. It is hoped that this will provide good guidance both for future research in the area, as well as for guiding estimates for delivery in practice.

8.1 Scalability in Practice

Our linking semantics was initially implemented as a centralized engine that loads all documents into memory and then evaluates the constraints. This has a number of advantages: architecturally, the engine imposes little infrastructure overhead, since it is represented by a single encapsulated component that is easy to deploy; the implementation of the engine itself is also straight-forward as it reads the documents into memory and evaluates the constraints there. For the majority of applications we have encountered, this approach is sufficient.

On the other hand, we have found specific applications that highlight the scalability limitations of such a centralized architecture:

1. The EJB case study presented in Section 7.2 exhibited long check times, mainly due to rule complexity. The check time had to be reduced in order to provide a more rapid response to developers.
2. We were presented with a requirement for validating a large number of derivative trades in a financial setting. The goal was to generate high throughput of single document validation.

3. We attempted to check several thousand newsfeed data files [Rivers-Moore, 2002]. The files themselves were small, but the huge number of files meant that it was again not possible to fit all of them into main memory at the same time to run a check.
4. A UML model that contained several thousand classes was too large to fit into volatile memory as a DOM tree.

These practical problems highlight two areas where scalability limitations are exposed: *run-time complexity* and *storage complexity.* As far as main memory usage is concerned, we can distinguish between two extremes: a small number of large documents, as in the case of the UML model, and a large number of small documents like the newsfeed data. It is important to address these extremes as they clearly occur in practice and cannot be handled by a centralized check engine.

Run-time complexity is mainly influenced by the complexity of the constraints and the number of documents. In many cases, such as the last two problems highlighted above, we can make use of information about the application domain in order to bring down check times. In the EJB case study it was clear that successive checks would be evaluating the same set of constraints on more or less the same set of documents. We may thus conjecture that checking only the constraints that apply to the changes made since the previous check, rather than the full set of constraints, can lead to time savings.

In the trade validation setting, we were presented with a large volume of *different* documents that all had to be checked independently against the same constraints. The most straightforward approach in this case was to replicate the centralized checking components so as to introduce parallelism, leading to higher throughput.

8.2 Analysis

Storage Complexity Storage complexity is affected by the number and size of documents. Since we are loading all documents into memory at the same time, the worst case complexity is the product of document number and average size:

Proposition 8.1. *Storage complexity for the centralized check engine is $O(d * n)$ in the worst case, where d is the number of documents and n the average number of nodes per document.*

Remark. Holds trivially if we take a node to be the storage unit.

If we hold either factor constant, storage complexity grows linearly with the other. In practice, storage complexity becomes relevant when the virtual memory available on the machine executing the check is not sufficient to hold all documents in a document set – we then have no way of carrying out a check.

Run-time Complexity Run-time complexity can be examined at varying levels of detail: for checking a whole set of rules, or for any single rule. For a set of rules, we will have to check all rules against all documents in the worst case:

Proposition 8.2. *Overall run-time complexity is $O(r*d)$ in the worst case for d documents and r rules.*

Remark. Follows trivially if we take a rule to be the unit of execution.

We can break up the factor r by looking at more detail for the run-time complexity of evaluating a single rule. The complexity of the rule itself, in terms of the number of nested subformulas, would seem to be a scalability dimension, but our case studies show that most constraints do not usually become more complex than a few nested quantifiers. The more interesting result is that informal experiments with profiling software on our implementation have shown that up to 90% of rule evaluation time is spent evaluating XPaths.

Path expressions exhibit runtime performance that decreases with the size of documents. We can thus expect longer check times as we add more, and bigger documents to a check. The *type* of path expression makes a difference too. Simple expression like `/Catalogue/Product` take slightly longer to evaluate on large documents, as every step in the path has to be compared to every node on the relevant level of the document tree, hence the complexity is a product of the number of steps and the breadth of the tree. More generic path expressions that require descendant traversal exhibit even worse behaviour. The expression `//*`, which selects all nodes in the tree, requires a full traversal and hence grows linearly with document size.

The check engine implementation already features path caching mechanisms to cut down on the number of evaluations that have to be performed, with good results. As far as scalability is concerned, however, we would like to be able to cut down on a very coarse grain level on the number of rules and paths that have to be evaluated – the next section is an example of how we can go about doing this.

8.3 Incremental Checking

Our case study in Section 7.2 has demonstrated that our checker can cope with software engineering documents in terms of expressiveness of the constraint language and in terms of the diagnostics it produces. We did however encounter problems with performance. While a check time of 23 seconds is not prohibitive, for reasons that we will discuss below, it can certainly be improved. The root cause of the problem is that the basic check engine always performs a full evaluation of all rules against all documents.

One strategy to improve performance is to only recheck those parts of documents relevant to changes made since the last check. To test whether this strategy can be applied in our checker, and to get some idea of the effort required to realise it, we have specified and implemented a simple incremental checking algorithm that supports a subset of XPath. The following is a discussion of our findings for applying the incremental checker in the context of the case study.

8.3.1 Background

The idea of incremental checking has been applied in the area of compilers and software development environments. Some software development environments take the idea to its extreme, aiming to execute an incremental check after every editing operation. It is important to distinguish upfront the goals and features of these environments from the goals and features of our checker so as to set a proper context for evaluation.

Most software development environments and modern CASE tools assume that the environment is under total control of the artefacts under modification. In practice, this often means that they are kept in memory and synchronized occasionally with central repositories. Essentially, these environments provide good speed of access and strict consistency guarantees by trading off some flexibility in editing.

By contrast, we assume that specifications are arbitrarily distributed and that editing and checking are separate processes - checks are then performed at the request of a user rather than automatically. Checks take longer to execute as documents have to be retrieved, but they can be executed flexibly. We envisage that developers will make several changes between checks, and that check times therefore do not have to meet timing constraints that are quite as strict as those required for an interactive system.

Another difference to note is that environments like eclipse, or research environments like ESF – see the discussion on page 21 for reference – implement *coded* static semantic checks. Event mechanisms provide triggers on document changes, and custom code is used to execute checks on abstract syntax trees. In our architecture, constraints are declarative and no matter whether we execute a full or incremental check, our algorithm has to analyse constraints. This makes our approach more generic, and makes it harder to specify an exhaustive algorithm. Since we execute checks at discrete time intervals, we also do not have access to the documents in memory – our checker is stateless between checks. This means that we cannot retain some of the caching information that these environments keep in memory in order to speed up incremental checks.

Against this background we will now discuss how incremental checking can be achieved in our architecture. This is followed by a discussion of the "path intersection" algorithms that determine rule applicability, and an evaluation in the context of the EJB case study.

8.3.2 Overview

When we execute a full consistency check, we check a document set against a rule set. Our incremental checker takes as additional input a linkbase – the result of a previous check – and a representation of the changes that have been made to any documents, as supplied by versioning systems. The output is a new linkbase that reflects the updated consistency status.

Our approach to incremental checking is to determine which rules need to be rechecked. This is a course grain approach because we are not attempting to recheck parts of a rule, but treat rules as a unit of execution. Our main reasons for proceeding in this way are that we do not need any additional information other than the rules themselves and a list of changes in the form of a diff, and we do not have to change the way rules are evaluated. This seems desirable as it limits the architectural overhead of realizing the incremental checking system.

Traditional versioning systems like CVS [Berliner and Polk, 2001] provide a textual difference computation between versions of documents. Similarly, document changes in XML documents can be computed using a tree difference algorithm [Tai, 1979]. We use our own algorithm, which provides linear time complexity at the expense of not always producing the minimum diff tree. Figure 8.1 shows the basic types of changes the algorithm detects: change of text content, change of attributes – attributes are not children of a DOM node in the strict sense, so they have to be treated differently from text content, – deletion of subtrees and addition of subtrees.

```
<?xml version="1.0" encoding="UTF-8"?>
<treediff>
    <changepcdata newvalue="Access Denied" oldvalue="Access Requested"
                xpath="/XMI/XMI.content[1]/..../Foundation.Core.Operation[1]/
                          Foundation.Core.ModelElement.name[1]"/>
    <changeattr xpath="/XMI/XMI.content[1]/..../Foundation.Core.Attribute[3]">
        <attribute name="xmi.id" newvalue="S.016" oldvalue="S.10016"/>
    </changeattr>
    <delsubtree xpath="/XMI/XMI.content[1]/..../Foundation.Core.Class[2]"/>
    <addsubtree xpathparent="/XMI/XMI.content[1]/..../
                              Foundation.Core.Namespace.ownedElement[1]"
              xpathleftsibling="/XMI/XMI.content[1]/..../
                                   Foundation.Core.Class[3]">
        <Foundation.Core.Class>
            ....
        </Foundation.Core.Class>
    </addsubtree>
</treediff>
```

Figure 8.1: Treediff output of changes to UML model

Given the documents, rules, changes and the previous linkbase, the incremental checker performs the following tasks:

1. "Intersect" the paths used in the rules with the paths in the tree difference – "diff paths" from here – so as to build up a list of rules that have to be rechecked.
2. Remove links that reference any selected rules from the old linkbase.
3. Run a check of the selected rules.
4. Insert the links created by the check into the linkbase and return the new linkbase.

The intersection of rule paths with diff paths is non-trivial, due to the complexities of the XPath language. The next section presents this concept in detail.

8.3.3 Intersection

In order to determine whether any particular rule needs to be re-checked, we perform a static analysis of overlaps between the paths in the rule, and all diff paths identifying change locations. We call this process "intersection". If any rule path intersects with a change location identified by a diff path, the rule needs to be rechecked. As we will see, what exactly is meant by "intersects" depends on whether a change represents a subtree addition to a document, a deletion, or a node value change. Before getting into this discussion, however, we need to discuss some assumptions.

The purpose of this study was to determine how a rule-level, coarse grain incremental checking algorithm would perform, and to determine its complexities. We thus chose a subset of path expressions that we wanted to permit in rules, and only one type of change – node value change – in order to get some results without getting into too much detail. The assumptions were chosen to cover just enough of XPath to meet the needs of the case study: by supporting the types of paths specified below we can cover all EJB profile, Java and deployment descriptor rules.

We considered the following subset of XPath:

- *Absolute paths* like `/ejb-jar/enterprise-beans` are permissible, including wildcard steps like `/ejb-jar/*/description`
- Absolute paths with descendant steps like `//enterprise-beans/description` are also permissible, but we do not allow wild cards
- No other XPath axes, like *ancestor* are allowed. Eliminating sideways and upward navigation greatly simplifies the discussion.
- No recurring element names are allowed in paths, for example in a hypothetical path `/x/y/z/y`, the element `y` occurs twice. The reason for this is rather involved and is explained in the pseudo-code implementation in 8.3.4. It is mentioned here for the sake of completeness.

- Position indicators are dropped. For example, /ejb-jar/enterprise-beans[2] would be simplified to /ejb-jar/enterprise-beans. This is perfectly permissible as the latter is a more generalized path of the former: any node the matches the first path would also match the second. Thus, when a rule points to /ejb-jar/enterprise-beans and a diff path identifies /ejb-jar/enterprise-beans[2] we will find a match, as expected.

With the context of a rule, some paths will be relative to variables, and hence do not meet the assumptions set out above. We pre-process paths like $x/description/text() by "flattening" them into absolute paths. To "flatten" variables, we look at the path used by the quantifier that introduced the variable and substitute it for the variable reference. This is applied recursively to produce a list of paths for a rule: for example, in the rule $\forall x$ ∈/ejb-jar($x/description/text()='Old') we would flatten the variable path to /ejb-jar/description/text().

The XPath language comes with its own predicate mechanism that permits the restriction of XPath expressions by testing properties of nodes during tree traversal. For example, /ejb-jar[description/text()='Old'] would select only those ejb-jar elements whose child description contains the text value Old. XPath specifies a boolean logic for use inside predicates, and permits the use of simple functions such as functions for string comparison. We extract any paths referenced by a predicate, be they contained in a function or directly as above, and append them to the containing step to which the predicate applies. Pre-processing the path above would yield two paths: /ejb-jar and /ejb-jar/description/text().

The last pre-processing step deals with simple parent navigation. Some of the rules in the EJB case study make use of a simple upward step, for example /java/class/../package. This type of path can be broken up into two separate paths for intersection: /java/class and /java/package.

With these assumptions out of the way, we will now sketch a simple approach to determining which rules apply, depending on whether an *addition*, *deletion* or *change* has occurred. For the purpose of this description we specify the *change* algorithm in detail in the next section.

We will take a quick look at the three types of document modifications in turn. Consider Figure 8.2, which shows a document tree after a new subtree has been added. The nodes n_5, n_6 and n_7 have been added. Any rule that accesses one of these nodes via a path, say $/n_1/n_3/n_5$, $/n_1/*/n_5$ or $//n_7$, will have to be rechecked. Rules that access other nodes, such as n_4 or n_3, will not be affected because no change has been made to these nodes.

To compute the intersection, we first have to create diff paths to all nodes in the added subtree – the diff algorithm supplies the addition location together with the nodes, but we

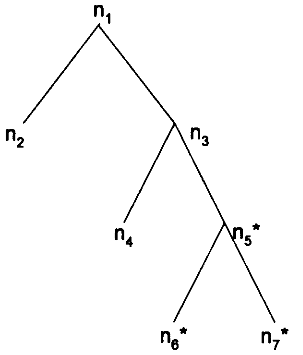


Figure 8.2: Illustration - subtree addition to document

need paths to all nodes. Then, to test a straightforward rule path like $/n_1/n_3/n_5$ we have to determine whether it matches a diff path exactly – all diff paths are absolute paths, and we check whether all steps match and the paths end with the same step. Wild card steps in rule paths match any diff path step. For descendant rule paths like $//n_3/n_5$ we have to scan forward in the diff path to see if n_3 is present and then continue the matching as normal with n_5.

Now consider the case of node deletion. Figure 8.3 shows a document tree where node n_5 has been marked as the root of a subtree to be deleted. The figure shows the document's state before deletion. In this case, any rules that potentially access nodes n_5, or the descendants n_6 and n_7 have to be rechecked. The problem is, the diff only tells us that n_5 has been deleted, not which subelements were deleted.

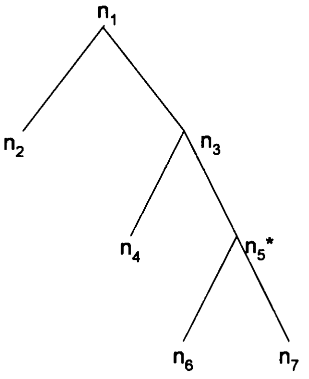


Figure 8.3: Illustration - subtree deletion from document

To determine whether deletion affects a rule path we could take the following approach: determine whether a rule path traverses to the deleted node – here, n_5 – or *beyond*. Rather than attempting to find a precise match, as in the case of addition, we would need to find out whether a path traverses *through* the change location. For example, the rule path $/n_1/n_3/n_5/n_7$ traverses through n_5, as does the path $//n_7$.

Descendant rule paths like $//n_7$ actually lead to an interesting problem with deletions: how are we to determine whether n_7 can potentially occur under a deleted node, if the only thing we know is the path to the deleted node itself, $/n_1/n_3/n_5$? The answer is, we would have to make use of the document's *grammar*, and such a grammar would have to be made available. It could tell us, using a depth first search, whether there is a potential path from n_1 to n_7 through the deleted node n_5.

Finally, consider the third case, node value change. This is illustrated in Figure 8.4. A node value change occurs when a text node or attribute node is changed – in the figure node n_4 has been changed.

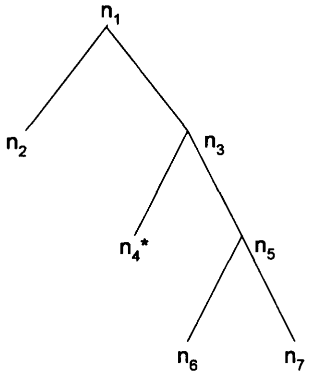


Figure 8.4: Illustration - node value change

With node value changes, we need to check all rule paths that access text or attribute nodes, that is, whose last step is `text()` or `@attr`, for any attribute `attr`. Similar to additions we attempt to match diff paths against all rule paths precisely. For example, `/ejb-jar/description/text()` would be accessed by rules that use exactly the same path, or by `/ejb-jar/*/text()`.

8.3.4 An Algorithm for Intersecting with Changes

After sketching the three algorithms, we proceeded to implement the algorithm for intersecting rules with document change diff paths. We chose this type of document modification, rather than addition or deletion, because it seemed to us to be the most straightforward to implement and give us a picture of the validity and limitations of our approach.

In this section we give a pseudo-code description of the intersection algorithm. The algorithm takes a list of node *change* locations and a list of paths from a rule, and returns a boolean indicating whether the rule needs to be rechecked. We will start by defining a simple structure for paths for use in the algorithm. This structure differs from the full definition of XPath given earlier in the thesis because it conforms to the limiting assumptions we have placed on paths in the previous section.

A step in a path is a tuple consisting of an axis, $Axis = \{child, descendant\}$, and a node name, which is a string: $Step = Axis \times String$. The string $*$ is a wildcard that can occur in paths in constraints, and matches any node name. A path is defined as an indexed list of steps.

Definition 8.3. $Axis = \{child, descendant\}$

Definition 8.4. $Step = Axis \times String$

Definition 8.5. $Path = [Step]$

The number of steps in a path p is denoted by its cardinality, $|p|$. For any path p, $p[i]$ is the i-th step of the path, the index ranging from 1 to $|p|$. The utility functions *axis* and *name*, given without specification, both take a step and return its axis and node name, respectively.

The intersection algorithm takes a list of paths from a rule, pre-processed as described in Section 8.3, and a list of diff paths. It returns $\top$ if the rule has to be rechecked.

Definition 8.6. does-rule-apply

```
function does-rule-apply
input  : rulepaths : $\wp(Path)$, diffpaths : $\wp(Path)$
output: $\top$ if any rule path intersects exactly with a diff path, $\bot$ otherwise

forall rulepath $\in$ rulepaths do
    forall diffpath $\in$ diffpaths do
        if intersect-exact(rulepath,diffpath) then return $\top$;
    end
end
return $\bot$;
```

The algorithm, *intersect-exact*, takes a diff path, which will be an absolute path containing only named node steps, and a rule path, which may contain descendant and wildcard steps, and returns $\top$ if they "match exactly". This means:

- We will consider each child step in the paths in turn, and ensure that they match by node name
- If a descendant axis occurs in the rule path, we scan forward to see if we find a matching node name somewhere in the diff path
- At the end of the algorithm, we need to have reached the end of both paths

Definition 8.7. intersect-exact

```
function intersect-exact
input  : rulepath : $Path$, diffpath : $Path$
output: $\top$ if $rulepath$ maches $diffpath$ exactly, $\bot$ otherwise

{Use two counters to mark the position in each path}
$rstep \leftarrow 1$;
$dstep \leftarrow 1$;

while $dstep \leq |diffpath|$ and $rstep \leq |rulepath|$ do
   if $axis(rulepath[rstep]) = child$ then
      if $name(rulepath[rstep]) = *$ or $name(rulepath[rstep]) = name(diffpath[dstep])$
      then
         $rstep \leftarrow rstep + 1$;
         $dstep \leftarrow dstep + 1$;
      else
         return $\bot$;
      end
   else if $axis(rulepath[rstep]) = descendant$ then
      {Scan forward in the diff path until we find a matching step}
      while $dstep \leq |diffpath|$ do
         if $name(rulepath[rstep]) = name(diffpath[dstep])$ then
            break ;
         else $dstep \leftarrow dstep + 1$;
      end

      {If we scanned over the end, there was no matching step}
      if $dstep = |diffpath| + 1$ then return $\bot$;

      {Otherwise, we can proceed to the next step}
      $rstep \leftarrow rstep + 1$;
      $dstep \leftarrow dstep + 1$;
   end
end

{Finally, return true if we reached the end of both paths}
return $dstep = |diffpath| + 1$ and $rstep = |rulepath| + 1$;
```

Here are some examples of paths that match according to this algorithm, diff path first in every case: `/a/b` matches `/a/b` because the paths are equivalent; `/a/b/c` matches `//b/c` because we can scan forward to `b` and from there on complete the path using simple steps; `/a/b` matches `/a/*` because a wildcard matches any named node step.

Examples of paths that do not match include: `/a/b` and `/a/c` because the second step has a different name; `/a/b` and `/a/b/c` because the rule path continues beyond the diff path; `/a/b` and `//c` because the element `c` does not occur in the diff path; and `/a/b/c` and `//b` because the diff path continues beyond the rule path.

We had previously mentioned a particular limitation on the paths that we support: no element names can be repeated along a diff path, for example `/x/y/z/y/u` repeats `y`. The reason is that if a descendant path of the form `//y/u` were processed against this diff path, the scanning algorithm would match the first `y` in the diff path and then report a mismatch of `z` against `u`.

This is not a fundamental problem, the limitation was chosen because a more complex algorithm was not necessary for the case study. The problem could be overcome by implementing a more complex algorithm that maintains a *stack* of diff paths, putting the original path on the stack by default. Then, when a descendant match such as `//y` is encountered, the algorithm could put duplicates onto the stack, storing a position counter with every possible match location, that is, the first or the second `y` in the diff path. A more thorough investigation would have to be performed if this approach was implemented in practice.

Complexity Overall, regardless of the type of change that is made, we get the following worst case behaviour:

Theorem 8.1. *For any particular rule, determining whether the rule applies takes in the time of $O(r * d)$ path comparisons in the worst case, where r is the number of paths in the rule and d is the number of diff paths.*

Proof. The worst case occurs when the rule does not have to be re-checked. It is clear from Definition 8.6 that for r rule paths and d diff paths, the algorithm will execute $r * d$ intersections before returning. □

Remark. If we were to take into consideration node deletions, the number of grammars needed to match descendant axes would also have to be taken into account. We have not investigated this, but can conjecture that with g different grammars we would obtain a worst case complexity of $O(r * d * g)$.

Path comparisons for node value changes are mainly linear in the worst case:

Theorem 8.2. *For a diff path with n_d steps and a rule path with n_r steps, intersection takes $O(n_d)$ comparisons in the worst case.*

Proof. Consider the algorithm in Definition 8.7. For any diff path with n_d steps, the first part of the outer loop condition ensures that it does not execute more than n_d times. □

We cannot comment further on the complexity of any intersection algorithm dealing with addition or deletion at this stage, this would require more investigation in further detail. It is likely that in the case of addition, adding a big subtree would cause problems with a naive approach, since we would have to create a diff path to every node in the subtree, followed by a comparison with all rule paths. Perhaps a more complex algorithm that implements a kind of depth first search traversal into the added subtree would be useful. For deletion, the size and number of grammars would have to be taken into account, since processing descendant axes would require a depth first search of grammars.

8.3.5 Evaluation

In order to determine the impact of the approach presented above, we have implemented the intersection algorithm for text and attribute changes. An initial implementation in Haskell was used to fine-tune the algorithm to flatten and intersect the different types of paths correctly, and was later ported to Java to process XML documents.

In our evaluation we wanted to find out how different changes affect rule selection, i.e. whether a small number of changes really leads to a small number of rules being selected, since this seems to be the key factor in determining whether any incremental checking approach at the rule level will be useful. This turns out to be difficult with our case study, since the number of UML rules is far greater than the number of Java and deployment descriptor rules. Furthermore, every single EJB profile rule over the UML model makes reference to a stereotype to test whether a UML class is an EJB class. Stereotype changes are therefore likely to be "high-impact" changes that require a lot of re-checking.

The number of changes in the diff file is also not as good a guide as the number of *types* of changes. For example, the path `/ejb-jar/enterprise-beans[1]` and the path `/ejb-jar/enterprise-beans[2]` are equivalent as far as the matching algorithm is concerned and are simplified into `/ejb-jar/enterprise-beans`. We have thus decided to classify types of changes as they would occur in a development life cycle with frequent re-checking: changes to Java code, changes to deployment descriptors and changes to the UML model. With each set of changes, we expect the intersection algorithm to select only the rules that are relevant to the type of artefact that is being changed.

We have labeled the sets of changes "small", "medium" and "large" or S, M and L. These are not intended as absolute values, or as indicating the impact we expect, but represent an approximation of the number of types of changes they contain. The sets are composed as follows:

- "Deployment (S)" contains two changes to home interface names
- "Deployment (M)" contains Deployment (S) and a change to the name of a bean – a high impact change as the UML-Deployment rules make use of that property
- "Java (S)" contains a change to the name of a class
- "Java (M)" contains Java (S) plus a change to a package name and the name of a variable in a class
- "UML (S)" contains a class name change
- "UML (M)" contains a class name change, an interface name change and a tagged value tag name change
- "UML (L)" contains UML (M), plus an operation name change and a stereotype name change

The fact that all changes are *name* changes does not make much of a difference - any node value could be changed.

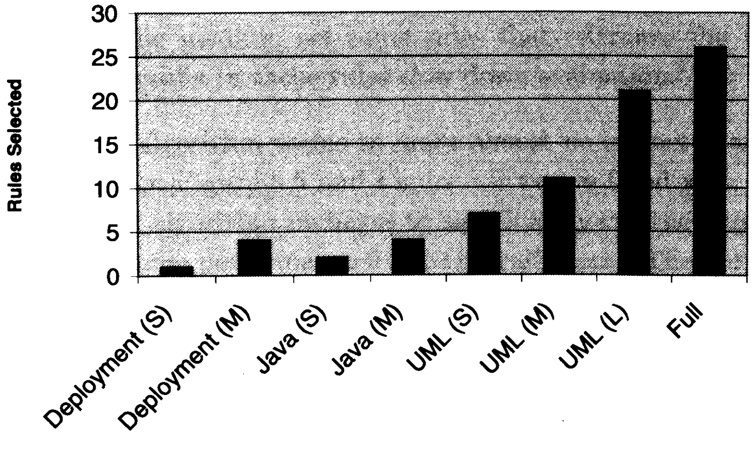


Figure 8.5: Number of rules selected for different change sets

Figure 8.5 shows the results of running our selection algorithm against the rules and the different sets of diffs, and the full set of rules on the rightmost bar for comparison. It meets our expectations in that changes to the types of documents that have a smaller number of rules, that is Java files and deployment descriptors, also result in a small number of rules being selected. For the UML models we get a progressively larger number of rules, reaching a full selection of all UML rules, though not the Java rules, as soon as the stereotype change is included in the change set.

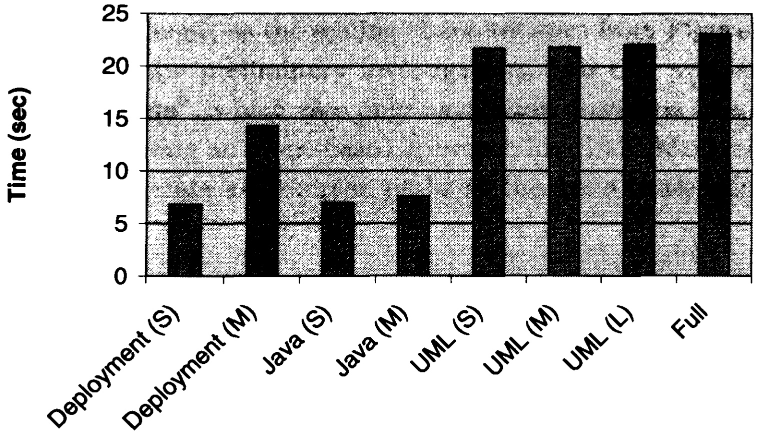


Figure 8.6: Check time for different change sets

While the algorithm seems to scale nicely in terms of the complexity of changes, the real world criterion for assessing the system as a whole must be the time taken for a re-check. Figure 8.6 shows the time taken for a re-check of each change set, and a comparison with the full check time. The checks were run three times on the same machine used in the earlier evaluation section, and an average value was taken as the result. For changes to Java files, we achieve a significant speed-up, bringing the total check time for the full check down to about 6 seconds from 23. The deployment descriptor check is also significantly improved, although for the medium set some rules that reference the UML model are chosen, and the complex paths in those rules slow down evaluation.

The only area where our algorithm seems to make almost no impact is where UML rules are selected. If we compare Figure 8.5 and Figure 8.6 we are faced with a problem: why is it that even though our algorithm manages to significantly cut the number of rules, we still get more or less the same performance for the overall check? The minor improvement that we get over the full check is simply the time that is taken up by the Java rules.

Almost all UML rules make reference to fairly complex paths initially, for example "`//Foundation.Core.Class`" to select all classes in the model. These paths take a very long time to evaluate – evaluating six complex paths, for finding all classes, stereotypes, dependencies and so on, takes significantly longer than evaluating the formulae. The reason is that the paths require a complete tree traversal of the rather large model tree to pick our their elements. While the selection for the small UML set selects only seven rules, these rules nevertheless include all six complex paths. The result is that the evaluation of the small set is only very slightly faster than for the medium set, and all sets are almost as slow as the full evaluation.

In essence, complex paths set a threshold for check times, and this is difficult to address – but it gives us a possible lead for future work: if we manage to bring down the evaluation time for paths that require full tree traversal, we can achieve a significant performance improvement, as the scaling characteristics from Figure 8.5 will kick in. We have already made some preliminary investigations into this problem. A path like "`//Foundation.Core.Class`", which uses only navigation down the tree, can actually be evaluated during parsing using an event-based approach like SAX [Megginson, 1998]. This approach would let us evaluate all complex paths at the same time, using only one tree traversal.

8.3.6 Conclusions

Despite the encouraging results in some cases, for example to changes in Java files, the overall conclusion from this chapter is that incremental checking at the rule level is complex – we covered only a subset of XPath for this study, but supporting full XPath may be quite difficult, – and not as effective as it should be. As soon as a stereotype name was changed, complex UML rules had to be rechecked in full, virtually eliminating any gains. As explained above, the check times in Figure 8.6 may be improved through clever path evaluation. Perhaps a more fine grained approach to incremental checking, which can recheck *parts* of a rule, is needed.

What made our approach to incremental checking tempting was that it required no changes to the architecture - no additional state information has to be saved. However, without additional information, more fine grained incremental checking will be difficult. If we associated further information with documents, we could "mark" all nodes that a rule touches. It might then be possible to recheck only nodes that have changed rather than the entire rule. Since we cannot modify documents stored in a distributed fashion, we would have to store this information on the machine executing the check in some serialized form, creating maintenance overhead. This is one of the key differences between our approach and a software development environment: if all artefacts are in memory and available as an abstract syntax graph it becomes much easier to store the required information for incremental checks.

It is also possible that some of the complexities with subtree addition and deletion, which we have not quantified in our investigation, may be easier to deal with once more detailed checking state is retained.

There is probably still some scope for our current approach to be applied in practice, though it would require some very specific conditions in order to be effective: if rules accessed quite different document locations, this would make it less likely that any given change selects a large number of rules; and it would probably suit scenarios with large numbers of rules, where rechecks are executed often, leading to a small proportion of rules being selected at any given time.

8.4 Replication

In some cases, what is required is not scalability with regards to growing document size or rule complexity, but simply raw speed. We performed a study on validating financial trade data that listed trade throughput as its main requirement.

The Financial Products Markup Language (FpML) [Gurden, 2001] is a markup language for financial derivative products. Since there was no satisfactory way of specifying static semantic constraints that check the validity of such products, we produced a list of constraints in our constraint language [Dui et al., 2003]. The rule set comprised 30 constraints and included rules that required plug-in operators for date calculations.

Validating a single FpML trade on a single, idle 1.7 GHz Intel-based machine against the 30 constraints took slightly over 300 msec, averaged over three runs. This check time is sufficient for processing about 300000 trade document in a 24 hour period. In some financial settings, where large amounts of trades are completed every day, this figure is not satisfactory.

The requirement of one particular system was to be able to process 600000 trades in 24 hours. The trades would have to be read off a message queue, and results be sent on to a delivery queue. Furthermore, it was a critical requirement that the trades were to be validated on inexpensive hardware, specifically Intel processor based machines rather than expensive server equipment – note that "expensive" in this setting refers to hardware architectures with more than two processors, which are not yet commoditized at the time of this writing – or mainframes.

Since the trade documents can be validated independently from each other, it was possible to replicate the "centralized" checking instances and let them work in parallel – note that this is not equivalent to the scenario in Section 8.5, where all documents have to be checked against *one another.*

8.4.1 Architecture

Figure 8.7 shows the architecture of our replicated checking system. Each replica is hosted on a single physical machine. The task of the replicas is to read trades off the document queue, check them and put the result linkbases on the result queue.

Each replica consists of two main components, a *message buffering layer* and a *checking layer.* The message buffering layer runs in a separate thread and polls the messaging

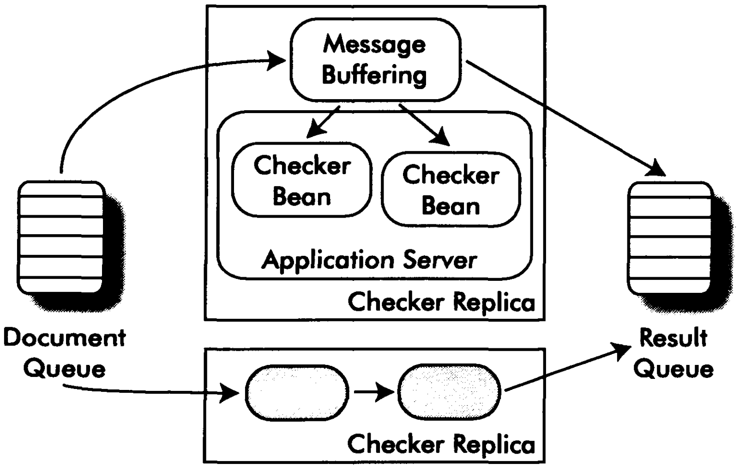


Figure 8.7: Replicated checker architecture

system for available trade messages. The checking layer executes as stateless session beans inside an EJB container.

Message Buffering Layer The message buffering layer decouples the checker from the messaging bus and tries to make sure that there are always messages available for the checking layer. The component initially pulls 20 messages off the queue, and if the number of messages in the buffer falls below 15, attempts to poll for more.

This layer was introduced after it became clear in preliminary trials that dequeuing and checking single messages introduced a performance bottleneck: if documents are retrieved individually between checks, the system is delayed by the transmission of the trade documents over the network. To get around this, we want to continue to pull documents off the network as much as possible while checks are executing in parallel.

Checking Layer The checking layer runs in a parallel thread of execution to the message buffering layer. Individual checking instances are hosted as stateless session beans inside a J2EE Application Server. There is no pressing need to run them inside a J2EE server, but this eliminated the overhead of writing further multi-threading code so as to take advantage of a dual-processor machine that we had at our disposal.

Ideally it would not have been necessary to separate the checking and message buffering layers, and we would have preferred to rely on the replication mechanism of an EJB container to replicate native Message-Driven Beans (MDBs). Unfortunately, we had no container at our disposal that implemented this replication facility. We also encountered some problems with implementing a system of replicated stateless session beans. While the container was able to replicate session beans transparently, the load balancing algorithms showed a tendency to starve slower machines so that scalability broke down after adding more than two machines to the systems. This led us to what is essentially a polling architecture, in which hosts download messages at a rate that is commensurate with their speed.

8.4.2 Evaluation

In our experiment, we wanted to achieve a target throughput rate of 600000 trades per day. Furthermore, we wanted to find out whether the architecture would accommodate a mixture of inexpensive hardware gracefully, in particular we wanted to accommodate different levels of processing powers in each machine. This was important as we did not have machines with exactly matching specifications at our disposal, and because in a real world setting it must be possible to add machines with better specifications when the throughput requirements increase.

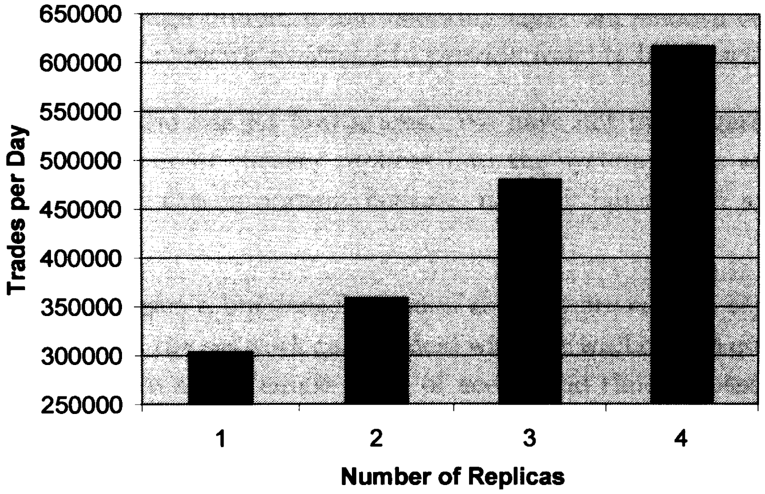


Figure 8.8: Document throughput vs. number of replicas

Figure 8.8 shows how the throughput of the system grows as replicas are added. Each experiment was performed using otherwise idle replicas. 2000 trades were randomly modified to eliminate caching by the message queuing system, and enqueued in the document queue. The time taken to clear the queue and terminate checking was measured and then extrapolated to calculate the per-day volume. Each experiment was repeated three times and the average time was taken as the result.

The first replica was a 600 Mhz laptop running Linux. The second was a badly configured SUN Ultra server, which checked fewer trades than the laptop and was thus suitable as a replacement for PC hardware. The third and fourth replicas were 1.7 Ghz Athlon PCs. The large difference in speed between the first two and last two replicas can be noticed in the super-linear increase in performance between the two and three-replica scenario.

With four replicas running we managed to check a volume equivalent to 617000 trades per day, which meets our evaluation target. The figure clearly shows that adding successive replicas led to a roughly linear increase in checking power. In particular, we can see that adding machines of different speeds does not in any way starve the slower replicas or throttle the faster ones. If either were the case, there would be no super-linear increase between replica two and three. Further evidence can be drawn from the fact that adding replica four, which has the same hardware specification as replica three, leads to an increase equivalent to that caused by adding replica three.

8.4.3 Conclusion

This case study demonstrates that it is possible to achieve a scalable checking architecture for checking large volumes of independent documents using relatively simple means. Our architecture with its message buffering and checking layer can smooth out differences in machine specification and network overhead to provide roughly linear scalability.

Clearly, this case study also has its limitations. We have not investigated the effects of adding a very large number of checker replicas into the system. The scalability of our architecture is limited by two important factors: network bandwidth and the message queues.

Network bandwidth becomes a bottleneck as soon as there are enough replicas retrieving documents so quickly that the network cannot deal with the traffic. The queueing systems, in quite a similar vain, are also a single point of access and thus a potential bottleneck. Since message queueing systems provide certain quality of service guarantees, there is a CPU overhead associated with enqueuing and dequeuing processes on the server side. When the number of replicas becomes too large the queuing servers get overwhelmed. This problem can be addressed by replicating the queues. Application servers like BEA WebLogic [BEA, 2004] provide facilities for doing this in a way that is transparent to queue receivers.

8.5 Distributed Checking

Memory scalability limits can sometimes be encountered when checking a large number of documents. In scenarios where a check is extended to many machines that hold multiple documents, a centralized checking service faces problems: the documents all have to be downloaded initially, even though the check is quite likely going to only access parts of the documents, and there may be too many documents to fit into the main memory of the checking machine.

In this section we discuss how to overcome this problem using a distributed checker – taking the check to the documents. We first discuss the architecture, design and implementation of the checker and then evaluate it in a case study where we checked a collection of up to 3000 newsfeed documents supplied by Reuters.

8.5.1 Requirements

Before embarking on the design of the architecture, we established several properties and features we wanted to put in place.

Middleware independence Since it was not altogether clear which type of middleware or which particular middleware implementation would offer the best performance, it was imperative that the architecture was flexible in this respect.

No distributed Document Object Model Most implementations of the DOM do not offer proper support for serialization. Serializing a node in a DOM tree would lead to the transfer of the whole tree over the network, since the tree is a fully connected data structure. This is clearly inefficient. Moreover, this approach would force us to rely on Java Remote Method Invocation (RMI) for object transfer, which was unlikely to provide the required performance.

Limited change impact At the time when we began this project, the centralized checking engine was taking the first steps on the road to becoming a commercial product. If the distributed checker was to be of any practical use in a product line, it had to be an *add-on* that customized the centralized checker, not a complete rewrite. The goal was not to touch the existing checking code at all and leave most of the internal architecture intact.

Low administration overhead Many distributed systems suffer from problems of practicality because of the sheer administration overhead they entail on each machine that is added to the system. We wanted to be able to provide lightweight components that can be executed standalone on client machines by specifying a list of documents for them to process.

The result is an architecture where documents are held by workers and do not need to be centrally loaded, and where a rather fine-grained communication mechanism is used to transfer node information during a check.

8.5.2 Architecture

In order to eliminate the necessity of transferring all documents to a centralized check engine, our architecture breaks down into the existing checking component, a "supervisor" component that forwards XPath expressions on behalf of the check engine and collects results, and "workers" that load documents and evaluate XPath expressions. The document set is partitioned between the workers either manually, as in our study, or could be assigned by the supervisor. Figure 8.9 shows an overview of the architecture.

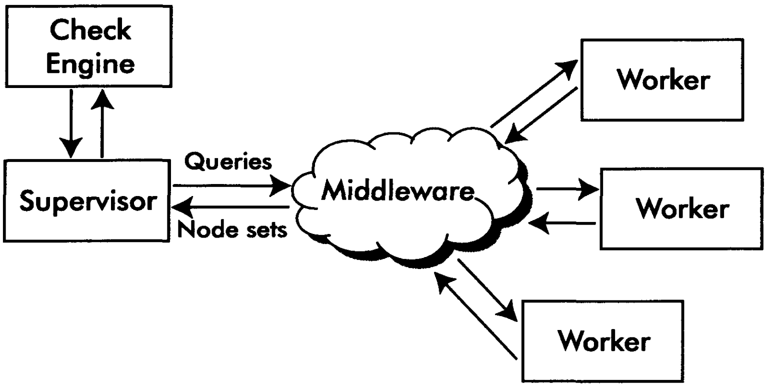


Figure 8.9: Distributed checker architecture overview

Within this architecture, the checking process comprises 4 phases:

1. **Worker Discovery** Before a check can be initiated, the workers are started on the hosts that will participate. The supervisor is then started, looks up the workers using a naming service and assigns a unique identifier to each worker.

2. **Document Loading** Once all workers have been registered, the supervisor issues a command to load the documents. Documents are loaded in parallel on all workers. The workers report back when they are ready to execute queries.

3. **Check** During the check, requests for XPath evaluation by the check engine are forwarded to the supervisor. Absolute expressions such as `/newsitem/text` are sent to all workers and the union of the results collected by the supervisor. Relative expressions like `$x/Name` need to be executed on the worker where the document node referenced by the variable `x` actually resides. The supervisor thus determines which worker to contact – we will see later how this is achieved – and sends a query to this worker only.

4. **Teardown** When the check is complete, the supervisor instructs the workers to unload their documents and the application terminates.

8.5.3 Design

When we refined our architecture into a more detailed design, we had to overcome two major problems: how to interface with the existing checking engine to distribute XPath evaluation without changing the existing code, and how to return results back from the workers without serializing DOM trees. We will illustrate our design through a discussion of these problems.

In the existing checking engine, each subformula is represented by an object. Those that require XPath evaluations, for example quantifiers, liaise with an XPath processor class. The result of an XPath evaluation is either a list of DOM nodes, or a primitive data type such as a string. This result is then interpreted by the formula.

For example, the quantifier formula $\forall x \in$ `/newsitem/text` (σ) would be evaluated as follows: the quantifier calls the `XPathProcessor` class to evaluate `/newsitem/text`. The processor class evaluates the expression over all DOM trees in memory, collates the results, and returns a list of DOM nodes. The quantifier class then iterates over this node list, binding the variable `x` to each node in turn, and evaluating σ for the current binding.

The checker abstracts from the underlying path processor using the Factory design pattern [Gamma et al., 1995]. Our distributed checker can thus register its own path processor. This processor does not perform any evaluation at all, but simply forwards the queries and binding context to the supervisor.

The second problem, reporting results back, is harder to overcome. Since path expressions are evaluated on the worker, we face the problem of having to send lists of DOM nodes back over the network to the supervisor and hence the check engine – a violation of our requirements.

We get around the problem by assigning unique identifiers to nodes that are selected in an XPath query. Each worker maintains two hash tables, mapping unique identifiers to nodes in trees and vice versa. When a query is complete, the worker sends back tuples of the form `(workerId,nodeId)`. The amount of data that has to be sent over the network is greatly reduced because the tuples contain only two long integers.

Since the existing check engine does not know how to treat nodes identified by unique identifiers, the supervisor has to masquerade the tuples behind the DOM `Node` interface. The result returned to the check engine then looks like a normal DOM node, and can be bound to variables in the binding context. These "fake" nodes can not be manipulated as freely as normal DOM nodes, but since the only manipulation is further path processing this is not a problem.

We will look at the evaluation of a relative path in order to clarify the interaction involved in an XPath evaluation. Figure 8.10 gives an overview of this process:

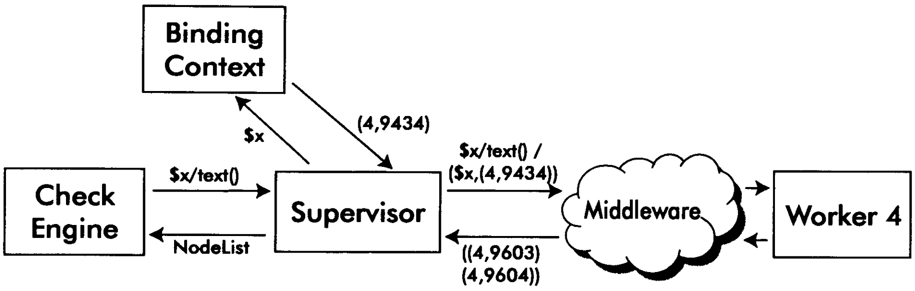


Figure 8.10: Distributed XPath evaluation

1. The check engine asks the XPath processor factory for an XPath processor. The factory returns a pointer to the supervisor.

2. The check engine sends the query **$x/text** to the supervisor, along with the current binding context.

3. The supervisor looks up the node pointed to by **$x**. This will be a "fake" node produced by a previous evaluation, so the supervisor inspects the tuple inside, which is **(4,9434)**. From this result the supervisor deduces that the node resides on the worker with identifier 4. It sends a query request containing the XPath expression and variable binding over the network.

4. Worker 4 receives the query. It resolves the node identifier **9434** using its hash map to obtain the actual node in the DOM tree. The worker then executes the path expression, which evaluates to two result nodes. The unique identifiers **9603** and **9604** are allocated for the two result nodes, and the result set **((4,9603),(4,9604))** is returned to the supervisor.

5. The supervisor wraps each tuple in a fake DOM node, and returns a node list to the check engine.

6. Formula evaluation proceeds in the existing check engine.

This design has a number of clear advantages, in particular it meets our requirement for leaving the existing check engine unchanged, and it sends only a very minimal amount of information over the network.

8.5.4 Implementation and Evaluation

In our case study we wanted to use the checker as a link generator to link related stories in newsfeed files. Figure 8.11 shows a fragment of a newsfeed file. We want to create links between all newsfeed items that have the same topic code listed for them. Figure 8.12 shows the rule that relates newsfeed files according to their topic code.

```
<newsitem itemid="804563" id="root"
         date="1997-08-17" xml:lang="en">
  <text>
   ....
  </text>
  <copyright>(c) Reuters Limited 1997</copyright>
  <metadata>
    <codes class="bip:topics:1.0">
      <code code="GCAT">
        <editdetail attribution="Reuters BIP Coding Group"
                    action="confirmed" date="1997-08-17"/>
      </code>
    </codes>
  </metadata>
</newsitem>
```

Figure 8.11: Newsfeed item fragment

```
<forall var="x" in="/newsitem/metadata/codes[@class='bip:topics:1.0']">
  <exists var="y" in="/newsitem/metadata/codes[@class='bip:topics:1.0']">
    <and>
      <not><same op1="$x" op2="$y"/></not>
      <equal op1="$x/code/@code" op2="$y/code/@code"/>
    </and>
  </exists>
</forall>
```

Figure 8.12: Newsfeed rule

We had a total of 3000 newsfeed files, each of which was 5kb in size. The files were distributed over a set of machines with varying specifications, ranging from SUN workstations with 128MB of memory to a dual-processor Intel Machine with 1GB. We initially selected JavaSpaces [Waldo, 1998] as a middleware as it had a simple interface and was straightforward to use. After running into problems with performance, we compared JavaSpaces to CORBA [Object Management Group, 2001] in a benchmark that simulated the type of messages we expected to send over the network. Since CORBA, using the OpenORB [exolab.org, 2001] Java implementation, was more than 20 times faster in this experiment, and was available on many platforms, we selected it as our middleware.

In order to assess the scalability of the architecture we tried to check sets of 500, 1750 and finally 3000 documents. We also varied the number of workers involved in a check from one to three and five. Two of the workers were general purpose PCs with 128MB of RAM, and the other three were shared servers with RAM varying between 384MB and 1GB. All experiments were repeated three times and the average times were taken as the result.

Table 8.1 shows the document loading time with a varying number of workers and varying numbers of documents. It was not possible to check 1750 or 3000 documents with one worker, and similarly 3000 documents with two workers, as both the physical memory

Workers / Docs	500	1750	3000
1	13987	-	-
3	8691	28756	-
5	5439	18369	61833

Table 8.1: Document loading times in milliseconds

limitations of the general purpose PCs and the shared memory availability of the servers prevented the checker from loading the documents. As is to be expected, the loading times decrease when the number of workers increases, as the workers load the documents in parallel.

Workers / Docs	500	1750	3000
1	10106	-	-
3	9705	80366	-
5	11338	81605	249650

Table 8.2: Check times in milliseconds

Table 8.2 shows the check times, without document loading, for a varying number of workers. The first thing to note about the numbers is that they do not seem to decrease with the number of workers. The reason for this is that the check engine spent most of its time evaluating the relative expressions like `$x/code/@code`, which have to be sent to a specific worker and thus cannot be evaluated in parallel. By contrast, the absolute expression in the quantifiers needs to be evaluated only once, rather than for each variable binding, and is evaluated fairly quickly.

The second point to observe is that the result for five workers seems to be slightly worse than for three, even though we expect it to be roughly the same by the explanation just given. The reason for this is that one of the workers that were added was a shared server that was running short on memory, causing it to start swapping during a check. The delay of one second should thus be regarded as a statistical error.

Finally, the figures clearly show that increasing the numbers of documents is expensive. The time per document increases from around 20 in the case of 500 documents to over 80 with 3000 documents. This increase is inherent in the quadratic complexity of the rule given in Figure 8.12, which performs a pairwise comparison of all documents.

We conclude our evaluation with a look at the network overhead encountered by the check engine. Table 8.3 shows the percentage of time the check engine spent waiting for network communication to complete as a portion of the total check time. The amount of time spent waiting decreases as the number of documents increases, because the quadratic complexity of the rule means that the check engine spends significantly more time in evaluation as

Workers / Docs	500	1750	3000
1	41.43%	-	-
3	38.03%	14.11%	-
5	46.76%	14.74%	15.70%

Table 8.3: Distribution overhead

the number of documents increases. The slight increase with five workers is again caused by the overloaded worker running out of memory.

8.5.5 Conclusion

This case study demonstrates an improvement in scalability in the number of documents, using a distributed architecture with fine grained communication. Using standard middleware technology and careful design we were able to distribute the checker with minimal impact on our implementation – the original checking code remained untouched throughout the study and distribution was kept transparent. By choosing CORBA, an industry standard, as our middleware, administration overhead on the workers could be kept relatively low: the CORBA libraries were packaged as part of the workers and no configuration was necessary on individual hosts.

We have also seen in our evaluation that there is obviously a penalty associated with distribution. A network overhead of up to 50% is the price to pay for the memory/speed trade-off provided by this distributed architecture. It is unclear at this point whether this architecture can truly scale to very large collections of documents. Increasing the collection of documents from 3000 to 300000, as would be required for validating news reports dating back several years, may not be feasible with this experimental implementation as the amount of network communication may lead to very long check times.

It would be interesting to try to exploit this architecture further by checking multiple rules in parallel. Our experiments have shown that the majority of XPath requests sent over the network are relative to a variable, which means that they are sent to only one worker. As a consequence, most workers are idle most of the time. By checking multiple rules at once, we could increase worker saturation.

8.6 Dealing With Very Large Documents

One area that we have not yet fully investigated is how to deal with very large documents: some documents are simply too large to be held in memory. The document object model adds a certain amount of overhead to the persistent representation of XML, which for our documents typically leads to an expansion of two to five times the document size. We have encountered an example of a document that could not fit into virtual memory in the area of software engineering.

In a case study on checking the Foundation/Core constraints of the Unified Modeling Language we were presented with an industrial model that was 27 MB in size. The model contained 2407 classes. Attempts to check the model on machines with 256 MB of RAM resulted in the check engine running out of memory.

Since it is not possible to hold such a large document in volatile memory, we have to relegate it to secondary storage. Some XML databases like Infonyte's PDOM [Infonyte, 2002] provide facilities for efficiently indexing, storing and accessing XML documents in secondary storage. Some of them provide a "cached" DOM tree that incrementally fetches portions of the document out of the database as the tree is navigated. Nodes that are not accessed frequently are discarded from main memory and refetched on the next database access.

By utilising an XML database with DOM tree support we could relegate documents to secondary storage when it becomes necessary without affecting the remainder of the check engine. Since the check engine and path processors operate on the DOM tree, the entire operation would remain transparent. All that is required is that the document is inserted into the database before the check.

We have done some preliminary work on an implementation of the database communication mechanisms. Initial results have been encouraging, showing that the check engine does not need to undergo extensive modification in order to support the alternative memory management mechanism. It was also possible to load the 27 MB UML document without encountering memory problems. However, we cannot at this point provide an estimate of the communication overhead encountered due to secondary storage access and communication with the database. We will aim to investigate this mechanism further and report on it at a later stage.

8.7 Chapter Summary

In this chapter we have examined the scalability factors of the check engine: rule complexity, the number of rules, the number of documents and document size. We have then investigated a number of options for addressing these factors.

Our case studies were centered around scenarios that called for quite different approaches: the fast rechecking times required by software engineering documents called for incremental rechecking, whereas memory limitations were better addressed by a distributed architecture that uses fine-grained communication to pass fragments of documents across the network, or by falling back to secondary storage. This shows that scalability of consistency checking is not a one-dimensional problem and that it is important to prepare in advance a number of solutions for addressing possible scalability problems.

This chapter also shows that it is mostly not necessary to come up with novel concepts to address scalability but that existing ideas can be adapted, albeit with some effort. Incremental checking is not a new idea and gained widespread acceptance in software development environments; secondary storage as a caching mechanism was also widely used in software development environments that used object databases to hold documents; and replication of components has become so widespread that it is a standard feature in Enterprise JavaBeans containers.

The different approaches explored here are independent: it would be relatively straightforward to combine an incremental checker with the distributed architecture if this should be deemed beneficial. We hope that we have aided this task by investigating each individual solution, and its intricacies, in detail.

The main conclusion that must be drawn from this chapter is that some scalability factors are more difficult to address than others. It was relatively straightforward to increase throughput of single documents using a replicated architecture - all we had to do was to make use of an application server and write some wrapper code. By contrast, creating a production quality incremental checking system would be difficult indeed: because consistency checks are executed individually and no state is retained in memory, determining which parts have to be rechecked is very difficult. Even when making limiting assumptions on the paths in rules, and concentrating just on rule selection, we had to go to some effort to establish a simple prototype. Further research in this area could perhaps concentrate on what auxiliary state information – maybe a list of all nodes "touched" by a path processor during the original check – could be saved alongside a linkbase to make this task easier.

This chapter, in the end, achieves the remit that we have set for it: we have identified the dimensions of scalability that can be addressed with straightforward solutions, and investigated others where we found architectural, or implementation difficulties. It is hoped that these investigations will act as guidance both for future work, and as a reference for further implementation in practice.

9 Related Work Revisited

In the light of the presentation of our contributions in the preceding chapters, we will now revisit the discussion of related work that we began in Chapter 2. In particular, it is now possible to point out in more detail interesting similarities and differences between the approach taken in our research and that in previous work.

There are many interesting parallels and similarly many differences between our approach and that taken in software engineering environments [Schäfer and Weber, 1989, Habermann and Notkin, 1986, GOODSTEP Team, 1994, IBM, 2003]. The syntax of the documents we manipulate is normally specified using an XML DTD or Schema, the SDEs use proprietary grammar notations; we construct DOM trees as an internal manipulation during a check, SDEs typically maintain an abstract syntax tree; and we generate hyperlinks between inconsistent elements where SDEs maintain a semantic graph by connecting syntax trees. All of this serves to illustrate that we are trying to steer clear of the proprietary mechanisms that these tools utilise, towards open and reusable standards. The advantages of this are clear: many tools already support these open standards, so integration is greatly facilitated and we do not face the monolithic task of wrapping tools and adapting them to a proprietary formalism [Emmerich, 1997]. Furthermore, we can make use of existing tools for processing documents and path expressions to achieve a more lightweight implementation. The second important difference is that SDEs build on centralised data structures. While our DOM trees are loaded onto a single machine, this is only necessary during the execution of the check – we do not force users to maintain their documents in a centralised fashion at all times.

Our work stands in the tradition of Viewpoints [Finkelstein et al., 1992] in a number of ways: the loosely coupled management of data, the tolerant approach to inconsistency, and the attempt to manage heterogeneous documents without tight integration. The notion of consistency rules [Easterbrook et al., 1994] between heterogeneous documents was also mentioned in this area. Although a lot of theoretical work on viewpoints and the associated consistency checking scheme exists, no generic implementation was ever provided. Early ideas on translating all specifications into a common representation like first order logic have never been shown to be practical. Our work combines the theoretical foundations combined with a lightweight and practical implementation on top of which a viewpoint framework could be established.

Research on federated databases is aimed at "enabling databases to cooperate in an autonomous fashion" [Ceri and Widom, 1993]. As data is distributed over many databases, these systems introduce mechanisms such as global queries and global constraint checking. Initial research in the area was built on very heavyweight mechanisms like global transactions [Simon and Valduriez, 1986]. Later work by [Grefen and Widom, 1997] and

[Grefen and Widom, 1996] is aimed at overcoming these requirements. By specifying a dedicated consistency manager on each host that is contacted by the underlying database when a transaction makes a change, and by establishing communication between the consistency managers through a constraint checking protocol, the need for global transactions is eliminated. Very elegant optimisations like the elimination of remote communication are possible by analysing the state before the commitment of a transaction and checking if the constraint requires a remote lookup [Gupta and Widom, 1993].

These approaches still differ significantly in their world view from what we are trying to achieve. They share our desire to validate distributed data on a global scale, but offer limited support for heterogeneity. Their view is that all data will be in relational databases, which somewhat facilitates the task of integration but makes it hard to check against proprietary tools and documents. They also assume that there will be at least local transaction processing available on each host, and that triggers will provide notification when changes are made by a transaction. In a loosely coupled scenario, documents may be stored simple as files on a file system, with no application in firm control. Furthermore, it may not be necessary to execute a check on every modification and we would prefer to leave the choice of process to the user.

Support for loosely couple distributed data is discussed in [Chawathe et al., 1996]. The authors recognise the need to adopt a tolerant approach to inconsistency as enforcement becomes too difficult in a distributed setting. The problem of heterogeneity is addressed by specifying an interface for manipulating data items on each host. Interfaces provide operations such as read, write or notify. We have chosen a similar approach, but we do not build on programmatic interfaces. Instead, we translate data into XML internally during a check, a more open and non-proprietary approach. Our work diverges in both its views on monitoring – we see monitoring as a separate issue – and the diagnostics we provide. While the paper correctly recognises accurate diagnosis as being of importance in the distributed setting, it makes no concrete suggestions for how feedback can be given to the user. By contrast, our linking semantics have been shown to be useful and precise in a number of case studies.

Traditional database integrity notions have been extended to cope with semistructured data [Buneman et al., 2000] and XML content in particular [Fan and Simeon, 2000]. The fundamental goal of this work and hence the approach is different. Like their predecessors in traditional database systems, these approaches focus on the prevention of inconsistency. They are also geared towards constraining single documents and feature no notion of distribution. The problem of verifying constraints on websites is discussed in [Fernandez et al., 1999] and applied in [Fernandez et al., 2000]. It is important to distinguish between the goals of these approaches and our own goals: Our constraints check if a set of data is consistent, whereas the approaches in the paper check if *any instance* of a schema graph will satisfy a given constraint. If the schema graph does not satisfy the property, modifications are suggested that will lead to valid instance graphs. Since we wish to tolerate inconsistency to introduce flexibility and because it is sometimes not possible to change the schemas of documents, for example when standardised schemas are used, we cannot adopt this approach but instead focus on detecting inconsistencies in instance documents.

Query languages have been used to specify constraints [Henrich and Däberitz, 1996]. In the context of XML, such an approach would be feasible by using an XML query language such as XQuery [Chamberlin et al., 2001]. We regard this as a lower level approach since this would in effect be asking the user to implement the checking semantics themselves instead of specifying a declarative constraint. There are also no distributed XML databases that could collaborate to answer a distributed query, as would be required by a multi-document constraint.

There are some parallels between our check engine and Schematron [Jelliffe, 2000]. Schematron is an XML validator that employs XPath and XSLT [Clark, 1999] to traverse documents and check constraints. It combines a boolean logic, namely that provided by XPath, with a simple reporting facility to provide a simple and elegant validation language. The overall focus of our work is however different from Schematron as we are aiming to support checks between multiple distributed documents. While such checks can be achieved in Schematron using XPath's `document` function, there would be no location transparency and the boolean logic would not be expressive enough to appropriately support semantic heterogeneity.

10 Future Work

We believe that this thesis addresses an interesting open problem and provides a solution that strikes a balance by being both practically motivated and implemented, and theoretically sound. This thesis necessarily presents only a snapshot of our research, and many avenues for investigation remain. We will thus give a brief outlook for areas of future research.

We have obtained many of our evaluation results from practical case studies in different application domains such as software engineering, financial data or news data. The need to be able to access legacy data sources in a seamless fashion through fetchers, the ability to be able to specify plug-in predicates, and the requirement for strong reporting mechanisms all appeared as the result of practical trials. We have also been fortunate to receive feedback from other research groups that used our tools for different purposes: for integration into a process engine to launch consistency checks automatically at different stages in a process [Cass and Osterweil, 2002], to check the semantics of KAOS [van Lamsweerde et al., 1991] requirements specifications or to check the semantics of the Little-JIL process definition language [Cass et al., 2000]. The feedback we have received from these projects has helped us to improve and stabilise the check engine. The tools are currently undergoing commercialisation to be sold into vertical markets and it is hoped that this process will yield further interesting requirements.

We are keen to try out the check engine in yet more areas. There are many application domains that could benefit from more rigorous consistency management, for example protein databases in bio-informatics are notoriously underconstrained, the financial industry has a need for checking inter-trade consistency between multiple trades and reference databases, and in software engineering research the problem of constraints on software architectures has recently appeared.

In our case studies we found that constraints were quite frequently not "equal citizens". It may be necessary to check one constraint as a precondition to another. For example, before we check that a value in one document correctly refers to a value in another document, we may want to check that this value uses the correct representation in the first place. We do not currently provide an automated dependency mechanism between constraints and it would be interesting to address this problem by annotating the constraints using metadata.

In certain business settings, the users equipped with the knowledge to be able to express constraints between documents are not necessarily XML literate – we have encountered this problem in practice in financial services, where business analysts expressed discomfort with having to express their constraints in XML. We are investigating options for providing a higher-level representation of our constraint language, perhaps one closer to natural language. Using an ontology that links business terms to their XML representation, we should be able to abstract from the underlying XML, and to later automatically compile constraints back into their executable representation. Such a compiler may also be useful for adding higher level constraints to CASE tools. For example, we could annotate models in UML editing tools, and later compile the constraints to execute over persisted XMI files.

We would like to integrate our checker directly into a number of tools in order to determine how expensive such an integration exercise would be. Several projects have already begun that will attempt to perform integration with CASE tools, and are performing pilot projects on integration into enterprise architectures.

Once a check has been executed and results have been provided, users may want to modify documents to eliminate inconsistency. In these cases, we may want to guide the users to ensure that any changes they make are effective: a change to a document that has nothing to do with a constraint violation will not eliminate the inconsistency. This act of guiding the user is what we broadly classify as document *repair*, and we have already made some inroads in this area. In [Nentwich et al., 2003a] we discuss how *repair actions* can be generated by specifying another denotational semantics over constraints. These repair actions, which are essentially parameterised document modification steps, can be shown to be both correct and complete, and thus give the user confidence and guidance in their modifications to documents. There are several technical and formal issues still to be overcome in this area, mostly caused by the complexity of XPath, and we will need to apply the approach in larger case studies before we can be confident of its practicality and usefulness. Nevertheless, this looks like a very exciting area for future research, and one that we will be keen to pursue.

11 Conclusions

This thesis addresses the problem of managing the consistency of distributed documents. We have overcome some of the limiting assumptions found in previous work, such as the need for integration or a central repository, the requirement for a heavy-weight transactional system, the programming rather than declarative specification of consistency constraints, and the insistence on eradicating inconsistency at the earliest possible opportunity. Instead, our approach pulls together other fruitful ideas from previous research in software engineering, most notably the tolerant treatment of inconsistency and the relation of heterogeneous views using consistency constraints. We have demonstrated that these concepts can indeed be implemented and deployed.

We have argued that there are benefits to be had from realizing consistency management through tool-independent services. This yields an encapsulated component that can be reused and is not tightly coupled to any particular tool. It also enables us to move away from proprietary formalisms that complicate integration and make it harder to address document heterogeneity. By building on a standard syntactic representation such as XML we overcome syntactic heterogeneity issues, and by utilizing a suitably expressive constraint language we can specify constraints that bridge the semantic gap between document types.

We hope that this thesis has also demonstrated that formalism and practical applicability are not as mutually exclusive as they are commonly perceived to be. We have used a denotational semantics to define a new interpretation of our constraint language with practical issues firmly in mind: we wanted to be able to point precisely to the elements that are involved in an inconsistency, and have done so by providing an interpretation that constructs hyperlinks between inconsistent elements. Since the denotational semantics essentially represents a simple mapping from the formula language, it was straightforward to implement.

In order for a thesis with roots in software engineering to become useful for practical purposes, the problem of scalability has to be considered. In our case studies with software engineering documents, financial data and news documents we encountered scalability problems among different dimensions of scalability. This thesis has made an attempt at a rough classification of the different dimensions of scalability, and at an analysis of how they affect run-time performance and storage requirements. We have seen that quite distinct techniques are necessary to address the different dimensions of scalability. We have been able to draw on a rich background from software engineering and database research to adapt existing ideas to our architecture, and in the course found some problem areas where this was difficult due to the inherent statelessness and flexibility of our approach. The scalability results presented here will certainly be useful as a means of quantifying the effort and cost involved in implementing different types of scalable architectures around the service model.

The formalisms and tools presented here make it possible for the first time to manage heterogeneous documents in a truly distributed fashion while retaining some control over their internal and inter-document consistency. We have since formed a company, Systemwire, that is taking these ideas forward into the commercial world. The most accurate test of the validity of our approach will perhaps be the success of this company.

A Wilbur's Bike Shop Documents

Sample advert

```
<Advert>
  <ProductName>ARO SHREDDER</ProductName>
  <Price currency="sterling">349.95</Price>
  <Description>Freestyle Bike. Super versatile frame for dirt,
  street, vert or flat. New full cromoly frame. Fusion MegaTube
  axle extenders.</Description>
</Advert>
```

Sample customer report

```
<CustomerReport>
  <CustomerIdentity>
    <FirstName>Licia</FirstName>
    <FamilyName>Capra</FamilyName>
    <Reg_Number>3645</Reg_Number>
  </CustomerIdentity>
  <Purchase>
    <ProductName>HARO SHREDDER</ProductName>
    <ProductCode>B001</ProductCode>
  </Purchase>
  <Purchase>
    <ProductName>Shimano LX Mountain Bike Crank Set</ProductName>
    <ProductCode>A102</ProductCode>
  </Purchase>
  <Purchase>
    <ProductName>Rock Shox Indy SL Suspension Fork</ProductName>
    <ProductCode>A107</ProductCode>
  </Purchase>
</CustomerReport>
```

Sample service report

```
<ServiceReport>
  <CustomerIdentity reg_number="3645"/>
  <Report>
    <ProductName>HARO SHREDDER</ProductName>
    <ProductCode>B001</ProductCode>
    <ProblemDescr>Found a problem in ...</ProblemDescr>
  </Report>
</ServiceReport>
```

Product catalogue fragment

```
<Catalogue>
  <Product>
    <Name>HARO SHREDDER</Name>
    <Code>B001</Code>
    <Price currency="sterling">349.95</Price>
    <Description>Freestyle Bike.</Description>
  </Product>
  <Product>
    <Name>HARO TR2.1</Name>
    <Code>B002</Code>
    <Price currency="sterling">179.95</Price>
    <Description>BMX / Trail Bike.</Description>
  </Product>
  ...
</Catalogue>
```

Consistency Constraints

```
<consistencyruleset>

  <globalset id="adverts" xpath="/Advert"/>
  <globalset id="products" xpath="/Catalogue/Product"/>

  <consistencyrule id="r1">
    <header>
      <description>
        The product name of an advertised product must be in the catalogue
      </description>
    </header>

    <forall var="a" in="$adverts">
      <exists var="p" in="$products">
        <equal op1="$a/ProductName/text()" op2="$p/Name/text()"/>
      </exists>
    </forall>
  </consistencyrule>

  <consistencyrule id="r2">
    <header>
      <description>
        All Adverts must match a  product in the catalogue with the same price
      </description>
    </header>

    <forall var="a" in="$adverts">
      <forall var="p" in="$products">
        <implies>
          <equal op1="$a/ProductName" op2="$p/Name"/>
          <equal op1="$a/Price" op2="$p/Price"/>
        </implies>
      </forall>
    </forall>
  </consistencyrule>

  <consistencyrule id="r3">
    <header>
      <description>
        All products listed as sold to a customer must be in the catalogue
      </description>
    </header>

    <forall var="p" in="/CustomerReport/Purchase">
      <exists var="c" in="$products">
        <and>
          <equal op1="$p/ProductCode/text()" op2="$c/Code/text()"/>
          <equal op1="$p/ProductName/text()" op2="$c/Name/text()"/>
        </and>
      </exists>
    </forall>
  </consistencyrule>

  <consistencyrule id="r4">
    <header>
      <description>
        All products for which a problem has been reported must have been sold to
        the customer who reported the problem.
      </description>
    </header>

    <forall var="s" in="/ServiceReport/Report">
      <exists var="c" in="/CustomerReport/Purchase">
        <and>
          <equal op1="$s/../CustomerIdentity/@reg_number"
                 op2="$c/../CustomerIdentity/Reg_Number/text()"/>
          <equal op1="$s/ProductCode/text()" op2="$c/ProductCode/text()"/>
        </and>
      </exists>
    </forall>
  </consistencyrule>
</consistencyruleset>
```

B Constraint Language XML Syntax

This appendix contains an XML Schema [Fallside, 2001] for the initial implementation of our constraint language in XML form. Our consistency check engine reads constraint files corresponding to this schema as its input.

The language contains all the constructs specified in the abstract syntax in this thesis, plus additional information for documenting constraints, controlling link generation and an additional predicate: the *operator* predicate provides call-outs to ECMAScript [ECMA, 1999] functions that take a number of nodes from DOM trees and return *true* or *false* as a result - an extension mechanism that can be used to perform complex computation or communicate with legacy systems.

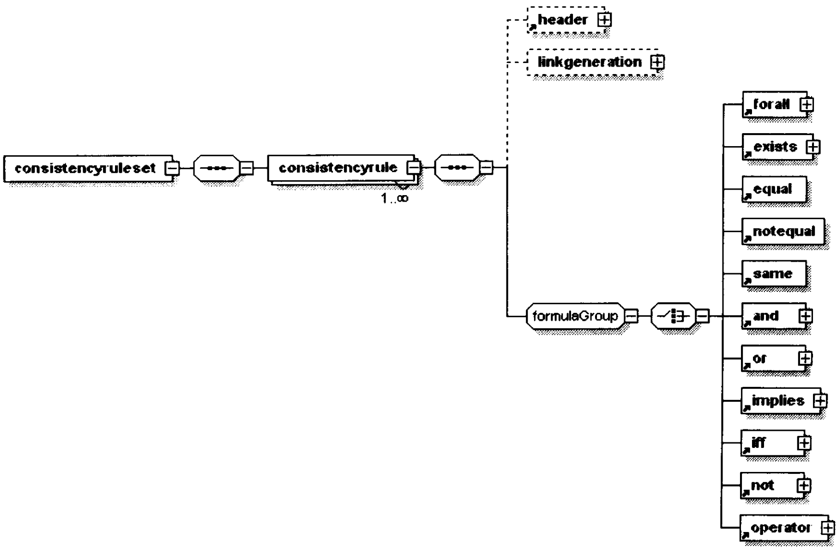


Figure B.1: Constraint Language XML Schema Overview

Figure B.1 is a graphical overview of the schema produced by an XML Schema editor. The following text is the XML representation of the schema in its standard form.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="http://www.xlinkit.com/ConsistencyRuleSet/5.0"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://www.xlinkit.com/ConsistencyRuleSet/5.0"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">
```

```
<xs:element name="consistencyruleset">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="consistencyrule" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="header" minOccurs="0" />
            <xs:element name="linkgeneration" minOccurs="0">
              <xs:complexType>
                <xs:choice maxOccurs="unbounded">
                  <xs:element name="consistent" minOccurs="0">
                    <xs:complexType>
                      <xs:attribute name="status" use="optional"
                      default="on">
                        <xs:simpleType>
                          <xs:restriction base="xs:NMTOKEN">
                            <xs:enumeration value="on" />
                            <xs:enumeration value="off" />
                          </xs:restriction>
                        </xs:simpleType>
                      </xs:attribute>
                    </xs:complexType>
                  </xs:element>
                  <xs:element name="inconsistent" minOccurs="0">
                    <xs:complexType>
                      <xs:attribute name="status" use="optional"
                      default="on">
                        <xs:simpleType>
                          <xs:restriction base="xs:NMTOKEN">
                            <xs:enumeration value="on" />
                            <xs:enumeration value="off" />
                          </xs:restriction>
                        </xs:simpleType>
                      </xs:attribute>
                    </xs:complexType>
                  </xs:element>
                </xs:choice>
              </xs:complexType>
            </xs:element>
            <xs:group ref="formulaGroup" />
          </xs:sequence>
          <xs:attribute name="id" type="xs:ID" use="required" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="header">
  <xs:complexType>
    <xs:sequence>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="author" type="xs:string" />
        <xs:element name="description">
          <xs:complexType mixed="true">
            <xs:sequence>
              <xs:any namespace="http://www.w3.org/1999/xhtml"
              processContents="skip" minOccurs="0"
              maxOccurs="unbounded" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="project" type="xs:string" />
        <xs:element name="comment" type="xs:string" />
        <xs:any namespace="http://www.xlinkit.com/Metadata/5.0"
        processContents="skip" />
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:complexType name="quantifierType">
  <xs:group ref="formulaGroup" minOccurs="0" />
  <xs:attribute name="var" type="xs:string" use="required" />
  <xs:attribute name="in" type="xs:string" use="required" />
  <xs:attribute name="mode" use="optional" default="exhaustive">
    <xs:simpleType>
      <xs:restriction base="xs:NMTOKEN">
        <xs:enumeration value="exhaustive" />
        <xs:enumeration value="instance" />
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:complexType>
<xs:complexType name="binOperatorType">
  <xs:group ref="formulaGroup" minOccurs="2" maxOccurs="2" />
```

```
</xs:complexType>
<xs:complexType name="binPredicateType">
  <xs:attribute name="op1" type="xs:string" use="required" />
  <xs:attribute name="op2" type="xs:string" use="required" />
</xs:complexType>
<xs:element name="forall" type="quantifierType" />
<xs:element name="and" type="binOperatorType" />
<xs:element name="or" type="binOperatorType" />
<xs:element name="implies" type="binOperatorType" />
<xs:element name="iff" type="binOperatorType" />
<xs:element name="not">
  <xs:complexType>
    <xs:group ref="formulaGroup" />
  </xs:complexType>
</xs:element>
<xs:element name="exists" type="quantifierType" />
<xs:element name="equal" type="binPredicateType" />
<xs:element name="notequal" type="binPredicateType" />
<xs:element name="same" type="binPredicateType" />
<xs:element name="operator">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="param" minOccurs="0"
      maxOccurs="unbounded">
        <xs:complexType>
          <xs:attribute name="name" type="xs:string"
          use="required" />
          <xs:attribute name="value" type="xs:string"
          use="required" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>
<xs:group name="formulaGroup">
  <xs:choice>
    <xs:element ref="forall" />
    <xs:element ref="exists" />
    <xs:element ref="equal" />
    <xs:element ref="notequal" />
    <xs:element ref="same" />
```

```
      <xs:element ref="and" />
      <xs:element ref="or" />
      <xs:element ref="implies" />
      <xs:element ref="iff" />
      <xs:element ref="not" />
      <xs:element ref="operator" />
    </xs:choice>
  </xs:group>
</xs:schema>
```

C Curriculum Case Study Rules

1. Every course must have a syllabus
2. The year of the course in the curriculum corresponds to the year in the syllabus
3. There must not be two courses with the same code
4. Each course listed as a pre-requisite in a syllabus must have a syllabus definition
5. A course cannot be a pre-requisite of itself
6. Every course in a studyplan is identified in the curriculum
7. A student cannot take the same course twice
8. 1st year BSc/CS and MSCi: 6 compulsory half-units must be present
9. 1st year BSc/CS and MSCi: 2 optional half-units must be present
10. 1st year BSc/CS and MSCi: no more than 1 optional half-unit can be non-programme

D EJB Case Study Data

D.1 UML Foundation/Core Constraints

D.1.1 Association

[1] The AssociationEnds must have a unique name within the Association
[2] At most one AssociationEnd may be an aggregation or composition
[3] If an Association has three or more AssociationEnds, then no AssociationEnd may be an aggregation or composition
[4] The connected Classifiers of the AssociationEnds should be included in the Namespace of the Association

D.1.2 AssociationClass

[1] The names of the AssociationEnds and the StructuralFeatures do not overlap
[2] An AssociationClass cannot be defined between itself and something else

D.1.3 AssociationEnd

[1] The Classifier of an AssociationEnd cannot be an Interface or a DataType if the association is navigable from that end
[2] An Instance may not belong by composition to more than one composite Instance

D.1.4 BehavioralFeature

[1] All parameters should have a unique name
[2] The type of the Parameters should be included in the Namespace of the Classifier

D.1.5 Class

[1] If a Class is concrete, all the Operations of the Class should have a realizing method in the full descriptor

D.1.6 Classifier

[2] No Attributes may have the same name within a Classifier
[3] No opposite AssociationEnds may have the same name within a Classifier
[4] The name of an Attribute may not be the same as the name of an opposite AssociationEnd or a ModelElement contained in the Classifier
[5] The name of an opposite AssociationEnd may not be the same as the name of an Attribute or ModelElement contained in the Classifier
[6] For each Operation in a specification realized by a Classifier, the Classifier must have a matching Operation

D.1.7 Component

[1] A Component may only contain other Components

D.1.8 Constraint

[1] A Constraint cannot be applied to itself

D.1.9 DataType

[1] A DataType can only contain Operations, which all must be queries
[2] A DataType cannot contain any other model elements

D.1.10 GeneralizableElement

[1] A root cannot have any Generalizations
[2] No GeneralizableElement can have a parent Generalization to an element which is a leaf
[4] The parent must be included in the namespace of the GeneralizableElement

D.1.11 Interface

[1] An Interface can only contain Operations
[2] An Interface cannot contain any ModelElements
[3] All Features defined in an Interface are public

D.1.12 Method

[1] If the realized Operation is a query, then so is the method
[2] The signature of the Method should be the same as the signature of the realized Operation
[3] The visibility of the Method should be the same as for the realized Operation

D.1.13 Namespace

[1] If a contained element, which is not an Association or Generalization has a name, then the name must be unique in the Namespace
[2] All Associations must have a unique combination of name and associated Classifiers in the Namespace

D.1.14 StructuralFeature

[1] The connected type should be included in the owner's Namespace

D.1.15 Type

[1] A Type may not have any methods
[2] The parent of a type must be a type

D.2 EJB Constraints

D.2.1 Design Model - External View Constraints

EJB Remote Interface

[1] The Class must specialize a model elements that represents the Java Interface, javax.ejb.EJBObject
[2] All of the Operations contained by the Class must represent EJB Remote Methods
[4] The Class must be the supplier of a UML Usage, stereotyped as <<instantiate>>, whose client represents the EJB Home Interface of the same EJB Enterprise Bean

EJB Home Interface

[1] The Class must specialize a model elements that represents the Java Interface, javax.ejb.EJBHome
[2] All of the Operations contained by the Class must represent EJB Home Methods
[4] The Class must be the client of a UML Usage, stereotyped as <<instantiate>>, whose supplier represents the EJB Remote Interface of the same EJB Enterprise Bean

EJB Session Home

[1] The Class must not be tagged as persistent
[2] The value of the EJBSessionType tag must be either Stateful or Stateless
[3] The Class may not contain any Operations that represent EJB Finder Methods
[4] The Class must contain at least one Operation that represents an EJB Create Method
[5] If the value of the EJBSessionType tag is Stateless, then the Class must contain exactly one Operation that represents an EJB Create Method. The type of its return Parameter must be the supplier of a UML Usage, stereotyped as <<instantiate>>, whose client is the Class

EJB Entity Home

[1] The Class must be tagged as persistent
[2] The Class must contain exactly one Operation that represents an EJB Primary Key Finder Method. The type of its return Parameter must be the supplier of a UML Usage, stereotyped as <<instantiate>>, whose client is the Class
[3] The Class must be the client of a UML Usage, stereotyped as <<EJBPrimaryKey>>, whose supplier represents an EJB Primary Key Class. The supplier must be the type of the in Parameter of the Operation that represents the EJB Primary Key Finder Method

EJB Primary Key Class

[2] The Class must contain implementations for Operations named hashCode and equals
[3] The Class must be the supplier of a UML Usage, stereotyped <<EJBPrimaryKey>>, whose client represents an EJB Entity Home

D.2.2 Design - Implementation Sample Checks

[1] Each remote interface declared in the UML model is implemented as a Java interface

extending EJBObject
[2] Each home interface declared in the UML model is implemented as a Java interface extending EJBHome
[3] Each bean class realizing a session bean home interface is implemented as a Java class implementing the SessionBean interface
[4] Each bean class realizing an entity bean home interface is implemented as a Java class implementing the EntityBean interface

D.2.3 Design - Deployment Information Sample Checks

[1] Each EJB Implementation Class declared in the UML model corresponds to an entry in the deployment descriptor
[2] If the name of an entity bean in the model matches a bean in the deployment descriptor, the bean in the deployment descriptor must be declared as an entity bean
[3] If the name of an session bean in the model matches a bean in the deployment descriptor, the bean in the deployment descriptor must be declared as an session bean

D.2.4 Implementation - Deployment Information Sample Checks

[1] Each bean listed in the deployment descriptor has an implementation for the given bean class, home interface and remote interface
[2] Each attribute listed as a container managed persistence field ('cmp-field') for an entity bean in the deployment descriptor must be an attribute of the bean implementation class

D.2.5 Implementation - Internal Checks

[1] For every remote interface there is a bean class that resides in the same package
[2] For every remote interface, there is a bean class in the same package that implements all the methods declared by the interface.

Bibliography

[Abramsky, 2004] Abramsky, S. (2004). Course on Game Semantics. http://web.comlab.ox.ac.uk/oucl/work/samson.abramsky/gsem/. Accessed January 2004.

[Altmann et al., 1988] Altmann, R. A., Hawke, A. N., and Marlin, C. D. (1988). An integrated programming environment based on multiple concurrent views. *Australian Computer Journal*, 20(2):65–72.

[Apache Software Foundation, 2003] Apache Software Foundation (2003). Xerces XML Parser. http://xml.apache.org.

[Apache Software Foundation, 2004] Apache Software Foundation (2004). Xalan XPath Processor. http://xml.apache.org/xalan-j/.

[Apparao et al., 1998] Apparao, V., Byrne, S., Champion, M., Isaacs, S., Jacobs, I., Hors, A. L., Nicol, G., Robie, J., Sutor, R., Wilson, C., and Wood, L. (1998). Document Object Model (DOM) Level 1 Specification. W3C Recommendation http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001, World Wide Web Consortium.

[Balzer, 1991] Balzer, R. (1991). Tolerating Inconsistency. In *Proceedings of the 13th International Conference on Software Engineering*, pages 158–165, Austin, TX USA. IEEE Computer Society Press.

[BEA, 2004] BEA (2004). WebLogic Platform. http://www.bea.com.

[Berliner and Polk, 2001] Berliner, B. and Polk, J. (2001). Concurrent Versions System. http://www.cvshome.org/.

[Borras et al., 1988] Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., and Pascual, V. (1988). CENTAUR: The System. *ACM SIGSOFT Software Engineering Notes*, 13(5):14–24. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, MA, USA.

[Bray et al., 2000] Bray, T., Paoli, J., Sperberg-McQueen, C. M., and Maler, E. (2000). Extensible Markup Language. Recommendation http://www.w3.org/TR/2000/REC-xml-20001006, World Wide Web Consortium.

[Buneman et al., 2000] Buneman, P., Fan, W., and Weinstein, S. (2000). Path Constraints in Semistructured Databases. *Journal of Computer and System Sciences*, 61(2):146–193.

[Cass and Osterweil, 2002] Cass, A. and Osterweil, L. J. (2002). Requirements-Based Design Guidance: A Process-Centered Consistency Management Approach. UM-CS-2002-024.

[Cass et al., 2000] Cass, A. G., Lerner, B. S., Sutton, S. M., McCall, E. K., Wise, A., and Osterweil, L. J. (2000). Little-JIL/Juliette: a process definition language and interpreter. In *Proc. of the 22^{nd} International Conference on Software Engineering*, pages 754–757, Limerick, Ireland. ACM Press.

[Ceri and Widom, 1993] Ceri, S. and Widom, J. (1993). Managing Semantic Heterogeneity with Production Rules and Persistent Queues. In *Proceedings of the 19th VLDB Conference*, pages 108–119, Dublin, Ireland.

[Chamberlin et al., 2001] Chamberlin, D., Florescu, D., Robie, J., Simeon, J., and Stefanescu, M. (2001). XQuery: A Query Language for XML. Working draft, World Wide Web Consortium (W3C). http://www.w3.org/TR/xquery/.

[Chawathe et al., 1996] Chawathe, S., Garcia-Molina, H., and Widom, J. (1996). A Toolkit for Constraint Management in Heterogeneous Information Systems. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 56–65.

[Clark, 1999] Clark, J. (1999). XSL Transformations (XSLT). Technical Report http://www.w3.org/TR/xslt, World Wide Web Consortium.

[Clark and DeRose, 1999] Clark, J. and DeRose, S. (1999). XML Path Language (XPath) Version 1.0. Recommendation http://www.w3.org/TR/1999/REC-xpath-19991116, World Wide Web Consortium.

[Clarke et al., 2000] Clarke, E. M., Grumberg, O., and Peled, D. A. (2000). *Model Checking.* MIT Press.

[Coward, 2001] Coward, D. (2001). *Java Servlet Specification.* Java Community Process JSR-000053, Version 2.3 edition.

[DeMichiel et al., 2001] DeMichiel, L. G., Yalcinalp, L. U., and Krishnan, S. (2001). Enterprise JavaBeans Specification v2.0. Technical report, Sun Microsystems.

[DeRose et al., 2001] DeRose, S., Maler, E., and Orchard, D. (2001). XML Linking Language (XLink) Version 1.0. W3C Recommendation http://www.w3.org/TR/xlink/, World Wide Web Consortium.

[Despeyroux, 1988] Despeyroux, T. (1988). TYPOL – A Framework to Implement Natural Semantics. Technical Report 94, INRIA, Roquencourt.

[Dui et al., 2003] Dui, D., Emmerich, W., Nentwich, C., and Thal, B. (2003). Consistency Checking of Financial Derivatives Transactions. *Objects, Components, Architectures, Services and Applications for a Networked World, Lecture Notes in Computer Science*, 2591.

[Easterbrook and Chechik, 2001] Easterbrook, S. and Chechik, M. (2001). A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, pages 411–420, Toronto, Canada. ACM Press.

[Easterbrook et al., 1994] Easterbrook, S., Finkelstein, A., Kramer, J., and Nuseibeh, B. (1994). Coordinating Distributed ViewPoints: The Anatomy of a Consistency Check. *Int. Journal of Concurrent Engineering: Research & Applications*, 2(3):209–222.

[ECMA, 1999] ECMA (1999). ECMAScript Language Specification. Language Specification 3rd Edition, European Computer Manufacturing Association.

[Eisenberg and Nickull, 2001] Eisenberg, B. and Nickull, D. (2001). ebXML Technical Architecture Specification. Technical report, Organization for the Advancement of Structured Information Standards (OASIS).

[Ellmer et al., 1999] Ellmer, E., Emmerich, W., Finkelstein, A., Smolko, D., and Zisman, A. (1999). Consistency Management of Distributed Documents using XML and Related Technologies. Research Note 99-94, University College London, Dept. of Computer Science.

[Emmerich, 1997] Emmerich, W. (1997). CORBA and ODBMSs in Viewpoint Development Environment Architectures. In Orlowska, M. and Zicari, R., editors, *Proc. of* 4^{th} *Int. Conf. on Object-Oriented Information Systems, Brisbane, Australia*, pages 347–360. Springer.

[Emmerich et al., 1995] Emmerich, W., Jahnke, J.-H., and Schäfer, W. (1995). Object Oriented Specification and Incremental Evaluation of Static Semantic Constraints. Technical Report 24, ESPRIT-III Project GOODSTEP, http://www.dbis.informatik.uni-frankfurt.de/REPORTS/GOODSTEP/GoodStepReport024.ps.gz.

[exolab.org, 2001] exolab.org (2001). OpenORB. http://openorb.exolab.org/.

[Fallside, 2001] Fallside, D. C. (2001). XML Schema Part 0: Primer. Recommendation http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/, World Wide Web Consortium.

[Fan and Simeon, 2000] Fan, W. and Simeon, J. (2000). Integrity Constraints for XML. In *Symposium on Principles of Database Systems*, pages 23–34.

[Fernandez et al., 1999] Fernandez, M., Florescu, D., Levy, A., and Suciu, D. (1999). Verifying Integrity Constraints on Web Sites. In *Proceedings of the 16th International Joint Conference on Articial Intelligence*, pages 614–619.

[Fernandez et al., 2000] Fernandez, M., Florescu, D., Levy, A., and Suciu, D. (2000). Declarative Specification of Web Sites with Strudel. *VLDB Journal*, 9(1):38–55.

[Finkelstein, 2000] Finkelstein, A. (2000). A Foolish Consistency: Technical Challenges in Consistency Management. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications (DEXA)*, pages 1–5, London, UK. Springer.

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

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

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Software*. Addison Wesley.

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

[Greenfield, 2001] Greenfield, J. (2001). UML/EJB Mapping Specification 1.0. Technical Report JSR-000026, Java Community Process.

[Grefen and Widom, 1997] Grefen, P. and Widom, J. (1997). Protocols for Integrity Constraint Checking in Federated Databases. *Distributed and Parallel Databases*, 5(4):327–355.

[Grefen and Widom, 1996] Grefen, P. W. P. J. and Widom, J. (1996). Integrity Constraint Checking in Federated Databases. In *Conference on Cooperative Information Systems*, pages 38–47.

[Group, 2002] Group, X. W. (2002). XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). Technical report, World Wide Web Consortium.

[Gupta and Widom, 1993] Gupta, A. and Widom, J. (1993). Local Verification of Global Integrity Constraints in Distributed Databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 49–58.

[Gurden, 2001] Gurden, G. (2001). FpML Version 1.0. http://www.fpml.org.

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

[Hapner et al., 1999] Hapner, M., Burridge, R., and Sharma, R. (1999). Java Message Service Specification. Technical report, Sun Microsystems, http://java.sun.com/products/jms.

[Henrich and Däberitz, 1996] Henrich, A. and Däberitz, D. (1996). Using a Query Language to State Consistency Constraints for Repositories. In *Database and Expert Systems Applications*, pages 59–68.

[Hintikka and Sandu, 1996] Hintikka, J. and Sandu, G. (1996). Game-theoretical Semantics. *Handbook of Logic and Language.* ed. J. van Benthem and A. ter Meulen, Elsevier.

[Hudson, 1999] Hudson, S. E. (1999). *CUP Parser Generator for Java.* http://www.cs.princeton.edu/ appel/modern/java/CUP/.

[Hunter and Nuseibeh, 1998] Hunter, A. and Nuseibeh, B. (1998). Managing Inconsistent Specifications: Reasoning, Analysis and Action. *ACM Transactions on Software Engineering and Methodology*, 7(4):335–367.

[Huth and Pradhan, 2001] Huth, M. and Pradhan, S. (2001). Model-Checking View-Based Partial Specification. *Electronic Notes in Computer Science*, 45.

[IBM, 2003] IBM (2003). The Eclipse Project. http://eclipse.org/.

[Infonyte, 2002] Infonyte (2002). *Infonyte PDOM.* Infonyte, http://www.infonyte.com/.

[Jackson, 2002] Jackson, D. (2002). Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290.

[Jelliffe, 2000] Jelliffe, R. (2000). The Schematron Assertion Language 1.5. Technical report, GeoTempo Inc.

[Knuth, 1968] Knuth, D. E. (1968). Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145.

[Marinescu, 2002] Marinescu, F. (2002). *EJB Design Patterns.* John Wiley & Sons.

[Matena and Hapner, 1999] Matena, V. and Hapner, M. (1999). Enterprise JavaBeans Specification v1.1. Technical report, Sun Microsystems.

[McWhirter, 2004] McWhirter, B. (2004). Jaxen XPath Processor. http://jaxen.org/.

[Megginson, 1998] Megginson, D. (1998). Simple API for XML.

[Nagl, 1996] Nagl, M., editor (1996). *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *Lecture Notes in Computer Science.* Springer Verlag.

[Nentwich et al., 2002] Nentwich, C., Capra, L., Emmerich, W., and Finkelstein, A. (2002). xlinkit: a Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185.

[Nentwich et al., 2001a] Nentwich, C., Emmerich, W., and Finkelstein, A. (2001a). Better Living with xlinkit. In *Proceedings of the 2nd International Workshop on Living with Inconsistency at ICSE 2001.*

[Nentwich et al., 2001b] Nentwich, C., Emmerich, W., and Finkelstein, A. (2001b). Static Consistency Checking for Distributed Specifications. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE)*, pages 115–124, Coronado Island, CA. IEEE Computer Science Press.

[Nentwich et al., 2003a] Nentwich, C., Emmerich, W., and Finkelstein, A. (2003a). Consistency Management with Repair Actions. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 455–464. ACM Press.

[Nentwich et al., 2003b] Nentwich, C., Emmerich, W., Finkelstein, A., and Ellmer, E. (2003b). Flexible Consistency Checking. *ACM Transactions on Software Engineering and Methology*, 12(1):28–63.

[Nuseibeh et al., 2000] Nuseibeh, B., Easterbrook, S., and Russo, A. (2000). Leveraging Inconsistency in Software Development. *IEEE Computer*, 33(4):24–29.

[Nuseibeh et al., 2001] Nuseibeh, B., Easterbrook, S., and Russo, A. (2001). Making Inconsistency Respectable in Software Development. *Journal of Systems and Software*, 56(11).

[Nuseibeh et al., 1993] Nuseibeh, B., Kramer, J., and Finkelstein, A. (1993). Expressing the Relationships Between Multiple Views in Requirements Specification. In *Proc. of the 15th Int. Conference on Software Engineering, Baltimore, USA*. IEEE Computer Society Press.

[Nuseibeh and Russo, 1998] Nuseibeh, B. and Russo, A. (1998). On the Consequences of Acting in the Presence of Inconsistency. In *Proceedings of the Ninth International Workshop on Software Specification and Design (IWSSD)*, pages 156–158. IEEE Computer Science Press.

[Object Management Group, 1997] Object Management Group (1997). *UML Semantics Guide.* 492 Old Connecticut Path, Framingham, Mass., ad/97-08-04 edition.

[Object Management Group, 2000a] Object Management Group (2000a). *The Meta Object Facility 1.3.* Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701, USA.

[Object Management Group, 2000b] Object Management Group (2000b). *UML Profile for CORBA Specification.*

[Object Management Group, 2001] Object Management Group (2001). *The Common Object Request Broker: Architecture and Specification Revision 2.5.* 492 Old Connecticut Path, Framingham, MA 01701, USA.

[OMG, 2000a] OMG (2000a). *Unified Modeling Language Specification.* Object Management Group.

[OMG, 2000b] OMG (2000b). *XML Metadata Interchange (XMI) Specification 1.1.* Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701, USA.

[Pietarinen, 2000] Pietarinen, A. (2000). *Games Logic Plays: Informational Independence in GameTheoretic Semantics.* PhD thesis, University of Sussex.

[Reps and Teitelbaum, 1984] Reps, T. W. and Teitelbaum, T. (1984). The Synthesizer Generator. *ACM SIGSOFT Software Engineering Notes*, 9(3):42–48. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA, USA.

[Rivers-Moore, 2002] Rivers-Moore, D. (2002). NewsML Version 1.1 Functional Specification. Technical report, International Press Telecommunications Council, http://www.newsml.org.

[Robins et al., 1999] Robins, J., Redmiles, D., and Hilbert, D. (1999). Argo/UML. http://www.ics.uci.edu/pub/arch/uml/.

[Schäfer and Weber, 1989] Schäfer, W. and Weber, H. (1989). European Software Factory Plan – The ESF-Profile. In Ng, P. A. and Yeh, R. T., editors, *Modern Software Engineering – Foundations and current perspectives*, chapter 22, pages 613–637. Van Nostrand Reinhold, NY, USA.

[Simon and Valduriez, 1986] Simon, E. and Valduriez, P. (1986). Integrity Control in Distributed Database Systems. In *Proceedings of the* 19^{th} *International Conference on System Sciences*, Hawai.

[Smolko, 2001] Smolko, D. (2001). *Software Agent Architecture for Consistency Checking of Distributed Documents.* PhD thesis, University of London.

[Tai, 1979] Tai, K. C. (1979). The Tree-to-Tree Correction Problem. *Journal of the ACM*, 26(3):422–433.

[Trancon y Widemann et al., 2001] Trancon y Widemann, B., Lepper, M., Wieland, J., and Pepper, P. (2001). Automized Generation of Typed Syntax Trees Via XML. In *Proc. of the XSE Workshop*, pages 20–23.

[van Lamsweerde et al., 1991] van Lamsweerde, A., Dardenne, A., Delcourt, B., and Dubisy, F. (1991). The KAOS Project: Knowledge Acquisition in Automated Specification of Software. In *Proceedings AAAI Spring Symposium Series*, pages 59–62. American Association for Artificial Intelligence.

[Wadler, 1999] Wadler, P. (1999). A formal semantics of patterns in XSLT. Markup Technologies.

[Waldo, 1998] Waldo, J. (1998). Javaspaces specification 1.0. Technical report, Sun Microsystems.

[Widom and Ceri, 1996] Widom, J. and Ceri, S., editors (1996). *Active Database Systems: Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann.

[Zave and Jackson, 1993] Zave, P. and Jackson, M. (1993). Conjunction as Composition. *Transactions on Software Engineering and Methodology*, 2(4):379–411. ACM Press.

[Zisman et al., 2000] Zisman, A., Emmerich, W., and Finkelstein, A. (2000). Using XML to Specify Consistency Rules for Distributed Documents. In *Proc. of the* 10^{th} *International Workshop on Software Specification And Design.* IEEE Computer Society Press. to appear.