USING HYPERLINKS FOR TRACEABILITY IN SOFTWARE ENGINEERING

By

GEIR A. BJUNE

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

December 2000

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of GEIR A. BJUNE find it satisfactory and recommend that it be accepted.

Chair

ACKNOWLEDGMENTS

I would like to thank Jack R. Hagemeister, my advisor, for his ideas and help with this project.

Another figure that has had tremendous value to this project is Tim Hoar, who has supplied the necessary information for getting the link traversal from Visual C++ to be possible.

This work would not have been possible without my parents, whose support really made a difference.

Last, but not least, Alethea, for being supporting throughout everything, reading and commenting on my work and being there to listen when I needed someone to listen to my complaints. Without her support, this project might never have been completed.

USING HYPERLINKS FOR TRACEABILITY

IN SOFTWARE ENGINEERING

Abstract

By Geir A. Bjune, M.Sc
Washington State University
December 2000

Chair: John C. Shovic

Traceability information in software engineering is often ignored and treated as if it were not particularly important. Part of this attitude can be blamed on the limitations of tools currently available in the marketplace. Many of the available tools are inflexible and difficult to learn.

HyperTrace, a low-impact tool that enables developers to insert and traverse hyperlinks between any information elements, resolves the issues it does so in a way that does not interfere with application use, but extends existing applications to provide traceability. HyperTrace is currently integrated in Visual C++ and Word, both commonly used applications from Microsoft. However, it can also be easily extended to new applications.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE

1 INTRODUCTION

Software built today can be distinguished from that built five years ago merely on the basis of size. The enormity and complexity of most current projects mean that specific information is not easy to find and the relevant specifications for a given piece of code can be difficult to track. Without prior knowledge of what a given piece of code is supposed to do, verifying that it is doing the correct thing, correctly, becomes extremely complex. When the number of documents produced during the course of a project it also considered, it is not difficult to imagine how problems might arise. Thus, project management becomes even more complicated, especially in situations where actual implementation status information is doubtful. The confusion created by such difficulties can delay product-shipping dates and may even have a detrimental effect on the quality of the shipped product. All these things seem unimportant in a general sense, since a software update can always be shipped or problems fixed in a later release, but with continuing use of embedded systems for more purposes than ever before, this argument might not be valid.

Traceability, generally defined in software engineering as the ability to describe and follow the life of a requirement both forwards and backwards from it's conception through development, refinement, and implementation gives us the solution to the problem of not being able to locate specifications for a given piece of code. Traceability is not a new concept, however industry seems to be lagging behind research in this area.

That does not mean that requirements traceability is not used at all and some organizations do employ requirements traceability throughout their projects [Dömges 1998]. In fact, organizations developing products for military or other government use, where strict rules regulate the process of developing software often employ traceability procedures.

One of the major problems with traceability is that there is not a single universally accepted definition. Traceability means many different things to many different people, complicates standardizing the process of gathering the required information.

Another problem facing the introduction of traceability into software engineering is that the classic view of it focuses on the traceability matrix, often included at the end of the document. The matrix may be cited as evidence that "we have traceability". However, developers often view this method of providing traceability as a boring, routine task that leads to nothing useful. Yet, lack of traceability can lead to lower work quality, since information that is relevant to the development and maintenance of the project is lost. Part of the problem is based in a lack of tool support for traceability. If the result of generating traceability information was something that could be more usable in the development and maintenance phases of the project, the status of the traceability information might be improved.

Furthermore, current tools that support traceability are often large and unmanageable. They are not flexible enough to match the process that was used in the original software development. Almost none of the tools commercially available today integrate into more than a select few tools from other companies. The goal of this project was to develop a traceability tool that could easily be integrated with nearly any other tool, in accordance with the notion that the tool should adapt to the process instead of the process to the tool.

The work described in this thesis includes an implementation of a hypertext based traceability tool, called HyperTrace. The tool is integrated into Visual C++ and Microsoft Word, but can also be used from web browsers, since the information needed can easily be embedded into HTML documents. By doing this, links can be provided between different information that will aid in document navigation.

The traceability information is easily described using hypertext, a collection of data items (nodes) connected through links. This approach gives the software engineer the ability to link between related information items and then traverse the established links by using a web-based point-and-click interface. Hyperlinks can be embedded into the source code, the documents (requirements document, design documents, test cases, defect logs and other related documents) or any other related piece of information.

One benefit of using hyperlinks is that doing so enables a common method of access to many kinds of information ranging from documents written in Microsoft Word, source code, published web pages and/or a query into a defect-tracking database. This approach goes well together with the shift towards putting more and more information on the web and making information web-accessible as well as enabling distributed environments. By using standard Uniform Resource Locators (URLs), links can be established to any type of resource placed either on the Internet or on a local network (intranet). It also enables placement of links to source code in any program that can traverse these hyperlinks.

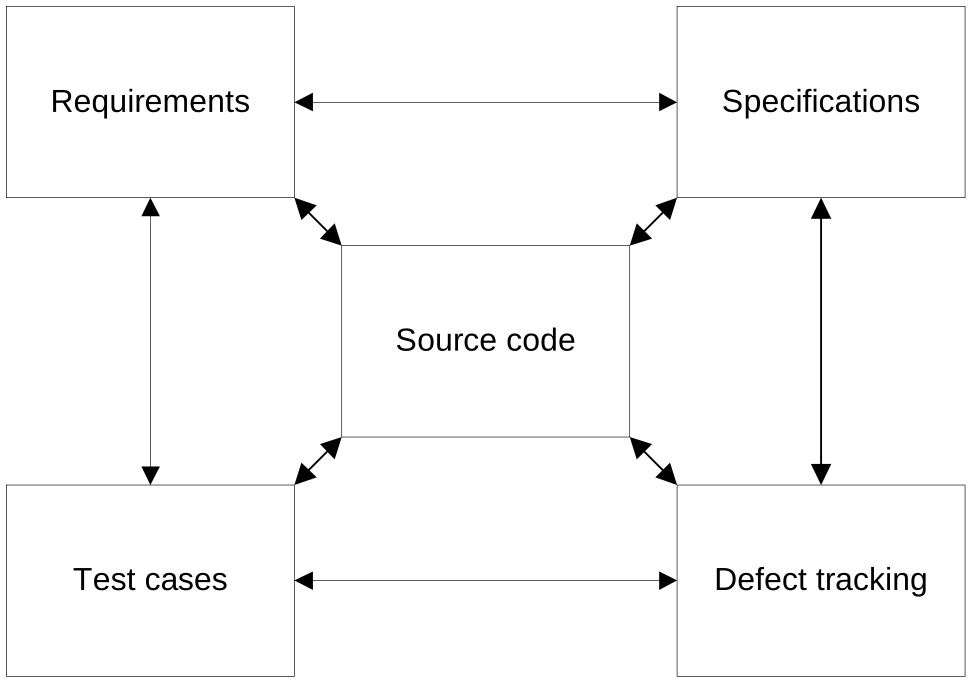


Figure 1.1: Document links

Figure 1.1 shows how documents can be connected using hyperlinks, allowing for both horizontal and vertical traceability.

The purpose of this tool is to increase the level of traceability used in software projects where it otherwise might not be a priority by allowing users to achieve easy insertion and traversal of links with a tool that is seamlessly integrated into the other tools they already use. The toolset resulting from this research aims to be low overhead, both administratively and in terms of demands placed on the computer on which it is run. Part of this is achieved by eliminating a database where everything is stored. Rather, hyperlinks are stored within the documents and files used by engineers. By showing the benefits of using requirements traceability and how easily it can be implemented within projects, may result in increased usage of traceability in software engineering projects.

The structure of this thesis is as follows: chapter 2 contains a discussion on the many definitions of traceability that can be found in literature, a comparison of traceability / requirements engineering tools available. Chapter 3 contains a description of the tool built for this project. Chapter 4 describes usage of the tools. Chapter 5 is the conclusion and a short discussion of future work.

CHAPTER TWO

2 RELATED WORK

Work related to this project includes research into hypertext/hypermedia, software engineering and traceability practices. Descriptions of some of the concepts and research models that have been previously developed are presented here

2.1. Hypertext/Hypermedia

Hypertext and hypermedia are two fairly similar concepts, diversified only by what is described, with hypermedia being more “media” oriented, and hypertext containing “just text”. Both concepts have as a central element the function “link”, which symbolizes a relationship between the two nodes it connects. A link in its simplest form can therefore be stored as a pair of nodes (source,destination). The process of linking is the creation of node pairs that will later be used to quickly locate related information.

Due to the versatility of hypertext, including its ability to easily describe and model relations between information elements, its concepts have been applied to many different areas. Being able to link related information elements provides a substantially more complete view of stored information and can be used to ease the location of related information regardless of the specific application.

One example of Hypertext’s ability to link easily between information nodes may be found in gIBIS [Conklin 1988] a policy/discussion tool wherein authors can enter information in nodes. Other authors can then comment on an existing node in new nodes, linked to the node on which they are commenting, with link types such as “supports”, “questions” and “responds-to”, to name a few. Discussions between authors thus become more easily readable, since the link semantics not only facilitate access to what is being said, but also gives the information “ownership” if a disagreement needs to be solved later.

Another early attempt at creating such hypertext-linking tool may be found in I-SHYS (Intelligent Software Hypertext System) [Garg 87], a hypertext-based tool for information management in software engineering. I-SHYS takes advantage of hypertext’s ability to store information of different types without changing the hypertext model. It is highly tuned towards software engineering with templates for standard software development processes. For data storage, I-SHYS uses an Ingres DBMS. All the information in the hypertext system is stored in this database, where it can be edited and viewed by multiple users, depending on the specific user’s role in the development project. I-SHYS users span a wide range of roles, from designer/implementer to end-user, each with its own particular goals. To support development processes, I-SHYS includes a set of tasks that should be performed, which are described using the waterfall model of software development.

2.1.1 General concepts

Hypertext has been applied to diverse problems for the very similar reasons. It provides the ability to navigate between information elements in a visual manner. It also hides the information type from the user, allowing different applications to be invoked depending on the type of link used, but still staying within the same semantic parameters

The most well known example of hypertext in use today is the World Wide Web. Most of the documents on the web are written in some form of HTML (HyperText Markup Language), which has become the standard for describing document structure. The links in HTML documents contain URLs, Uniform Resource Locators. These URLs describe where the information can be found and how to access it. A URL is divided into three parts. First is the protocol to be used to access the data. The second part is the server at which the data can be found and third is the information the server needs to locate particular data. An example URL: http://www.eecs.wsu.edu/index.html, tells the program (or user) that in order to access the referenced information it must use the "http" protocol (hypertext transfer protocol) to connect to "www.eecs.wsu.edu" and ask for the document "index.html". This example is simplified, since the protocol portion can be substantially more complicated, containing authentication information needed to access the data, which for ftp is "ftp://<user>:<password>@server". Another form of URL is the well-known e-mail address link "mailto:<user>@<server>.

2.1.2 Making tools hypermedia-aware

For tools to support hypertext or hypermedia, they must possess built-in support or a workaround must be created. According to [Whitehead 1997], the three primary ways of integrating applications into hypermedia systems are: launch-only, wrapper and customized integration. Each of these offers a different level of hypermedia support for which the costs associated with development also vary. For example, custom integration implies modification to the original application, which offers the most in terms of benefits, but also costs most to implement, whereas launch-only is less expensive, but functionally limited.

“Launch-only” is described as having the ability to fire up an application from a link traversal, but not the ability to specify where in the application/document the user wants to go. The ability to open the correct file is helpful, but severely limited, since there can be no hyper-linking to specific locations within a document opened by the application. Figure 2.1 shows the launch-only integration between HyperDisco, a hypermedia system, and XEmacs.

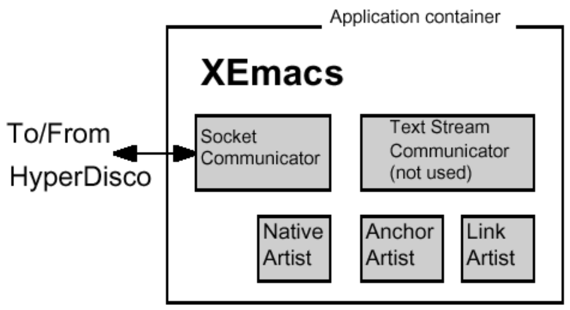


Figure 2.1: Launch-only integration [Whitehead 1997]

The second integration style described by Whitehead is “wrapper,” which uses a separate piece of software to translate from a one way of communicating hypermedia events (e.g. Chimera) to another (e.g. Microsoft Word style hyperlinks). Via this protocol, Microsoft Word could participate in a Chimera hypermedia system. Figure 2.2 shows Chimera and FrameMaker integration using a wrapper.

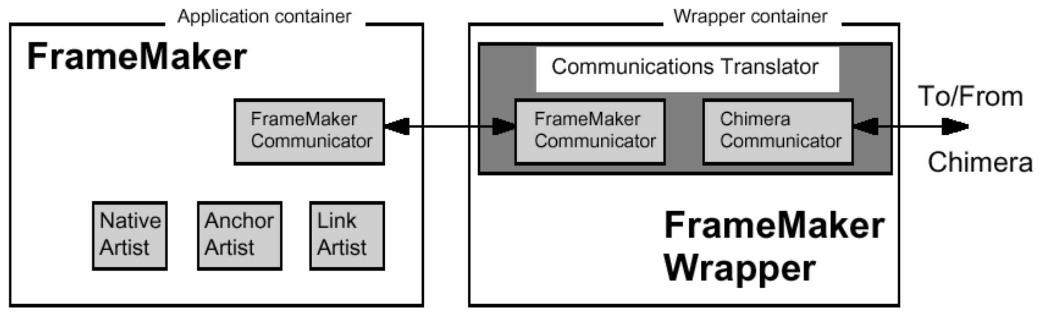


Figure 2.2: Using wrappers to integrate tools [Whitehead 1997]

The third style of application integration is "custom", consisting of modifying the application behavior directly, either through a customization language or through source code. This is used if the application to be integrated has no direct hypermedia concept, but can be extended. The custom integration is often the most complicated to complete, but also offers the most flexibility, since the user controls the entire integration process (if the application can be extended enough).

2.2. Open Hypermedia Systems (OHS)

The Open Hypermedia Systems Working Group is currently engaged in expanding the use of open hypermedia systems by developing new and better concepts. Most research in open hypermedia systems has remained at the research stage without ever being widely adopted, a situation OHS hopes to change with the Open Hypermedia Protocol. The goal of the OHSWG is to create a hypermedia system similar to that of the web, but with centralized servers and applications that are more hypermedia-aware, making the links and underlying hypermedia architecture less visible to the user. Such a move would standardize how applications interact with hypermedia in a way different from that of the web-centric world.

2.2.1 Open Hypermedia Protocol (OHP)

Described in a working draft [Davis 1997] from the Open Hypermedia Systems Working Group (OHSWG), the OHP can be distinguished from html in that the links are not embedded in the documents, but held separately from them. In OHP, the servers hold the links and anchors and the locations of links are specified in meta-data, nodes. The nodes function as wrappers for the documents, providing further distinction between the two. However, since OHP is still a work in progress, little is known about what it will be able to offer or what impact that will have on the hypermedia world.

2.3. Chimera

Chimera [Anderson 1994] is an open hypermedia system designed to help describe and maintain the collection of software objects in a software development environment. It uses different viewers (including browsers and editors of the software objects).

Chimera features its own protocol based on using RPC communication between the viewers and the server. In this way, the viewers are able to generate hypermedia events, and receive them from the server as well, with message passing capabilities implemented via RPC. Chimera also offers an API for extending applications to support their hypermedia events, making the task of implementing Chimera aware applications easier. Applications would then register for these events and act accordingly if an event is triggered in the Chimera server. Chimera stores the hyper-web (the links database) in XML on the Chimera server.

Figure 2.3 shows the conceptual architecture of Chimera and how it uses communication between the viewers and the centralized Chimera database. As shown, the server can invoke viewers based on the type of document the client requests, while the client stores information about the viewers locally. It also allows interface with external systems as shown.

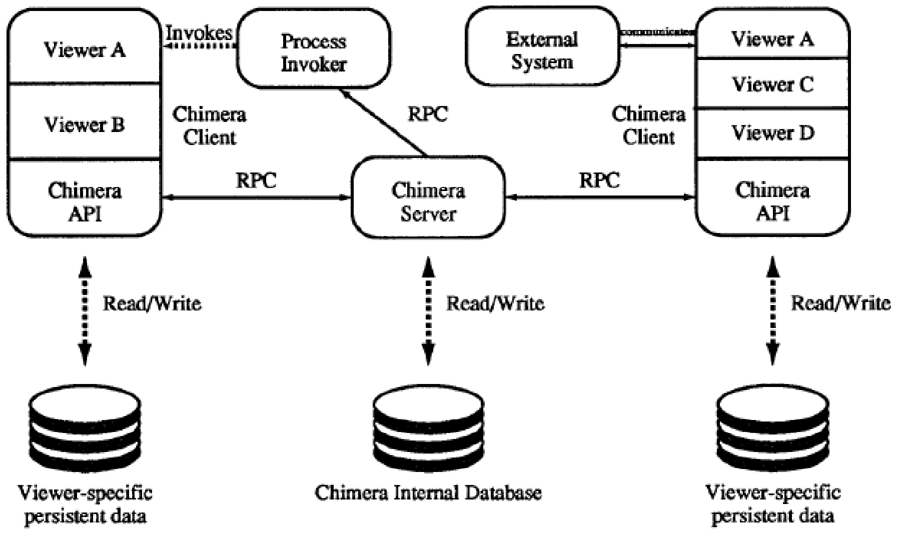


Figure 2.3: Overview of Chimera conceptual architecture [Anderson 1994]

One of the viewers Chimera uses is FrameMaker 5 (see figure 2.2 for wrapper architecture), which supports hypermedia events through a wrapper built to map the Chimera events into events that FrameMaker handles. Other applications, like a clone of the Unix® editor vi were also extended to support Chimera, thus showing the possibility for extending applications to support Chimera as well as other socket based environments. XEmacs has also been integrated with Chimera, as described in [Whitehead 1997].

Chimera has been continuously updated, the latest version being 2.0 [Chimera]. This version supports integration with XEmacs (a set of elisp macros) and FrameMaker 5 on both Solaris and Windows. The latest version also has a COM client, which allows integration with any application that can use COM, either through development of a new application version or through macros or other forms of extending the program (as can be done with Microsoft Office, Microsoft Visual Studio and many more).

However, the Chimera system, at least in its earliest versions, was not very scalable. [Anderson 1999]. This was primarily caused by the fact that the server stores the link information and therefore needs to maintain a database (the hyper-web stored in XML). The project used for testing was an XML file containing 500,000 links, which, when parsed into an XML parse tree, became too large to keep in memory. This occurred primarily because the entire XML tree was parsed at once, but can also be attributed to the fact that Chimera utilizes a central server that stores information rather than enabling clients to parse the XML when needed. By redesigning around a different XML parser engine, Chimera developers showed considerable performance gains, however, parts of the server system had to be redesigned to remove the database from memory.

As noted in [Anderson 1999], problems resulted from trying to integrate the World Wide Web into Chimera. One of the perceived limitations is that the World Wide Web is limited to HTML documents, which means that documents have to be converted into HTML documents, where they cannot easily be edited. Such difficulties are avoidable by converting the document to HTML, but always maintaining the original document and again converting to HTML after updating. This limitation is no longer relevant for most of today's platforms, since most current applications (like the Microsoft Office 2000 application suite) readily support links of the type used on the World Wide Web, thanks to the Web's widespread user acceptance.

2.3.1 Summary of hypertext and hypermedia

The latest in hypermedia research intersects the research done for this project in many ways. The Open Hypermedia Protocol style of implementing link storage and traversal looks interesting but has some drawbacks, especially in the use of a central database in which to store links. OHP may be good candidate for integration provided it becomes successful. However, part of the rationale for avoiding a central database lies in the results from [Anderson 1999], where scalability problems arose. Even if those particular problems were resolved, it is probable that a more effective long-term scalability solution is to be found by avoiding a central database altogether.

2.4. Traceability

To understand traceability in a software development context, we must first understand what the word "traceability" means. Traceability can be defined as the set of relationships that exist between information elements in a software engineering project. An example relationship could be the one that exists between a piece of source code and a test case that exercises that software.

In the Webster dictionary traceability is defined as follows:

To follow the footprints, track, or trail of

To follow or study out in detail or step by step <trace the history of the labor movement>

To discover by going backward over the evidence step by step <trace your ancestry>

To discover signs, evidence, or remains of <something>

If the existence of a link between two nodes is used to model a relationship, hypermedia provides a very flexible set of ways by which traceability can be captured.

Another definition, more closely related to software development comes from the IEEE Standard Glossary of Software Terminology [IEEEGST]:

The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match.

The degree to which each element in a software development product establishes its reason for existing; for example, the degree to which each element in a bubble chart references the requirement that it satisfies.

2.4.1 Different forms of traceability

Traceability can, given the rich set of definitions in literature, be used for many different things. The two main ways traceability definitions can be grouped is horizontal and vertical traceability. Both forms have advantages, based on what information needs to be captured.

2.4.2 Horizontal Traceability

Horizontal traceability is defined as traceability information pointing to products of a successor or predecessor phase in the software development life cycle. The information gained through horizontal traceability can be used to get an overview of the causal effect of previous information elements. Figure 2.4 illustrates horizontal traceability between a set of documents (simple case).

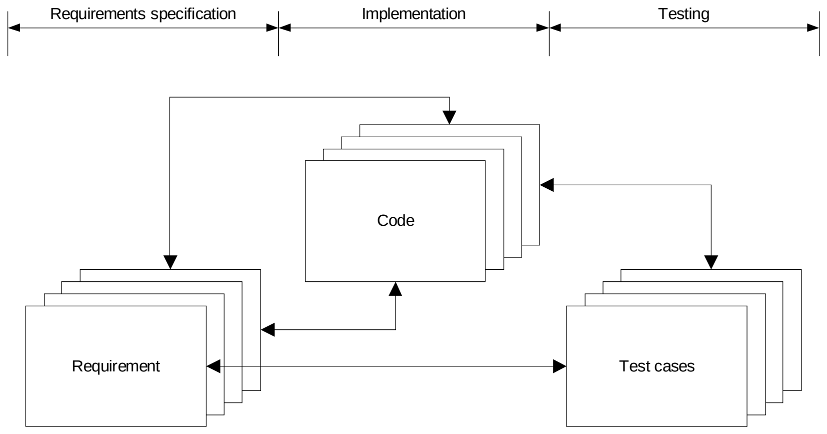


Figure 2.4: Cross-phase relationships

Different variations of this form exist, most of which vary in the amount of traceability information gathered and the particular development process phases from which information can be traced, Variations range from the simplest form, tracing requirements to implementation, to models tracing information from every phase to the phases to come (and sometimes even back to the previous phases). Horizontal traceability is often used to establish whether the output of one phase is what it should have been, given input to the phase.

2.4.3 Seven different definitions of horizontal traceability:

To illustrate the varying definitions of traceability, we have shown seven different definitions all found in current literature, related to horizontal traceability. Noteworthy is the lack of one common, unifying definition, although they are all similar.

Mil-std 498 [MILSTD498] specifies that traceability information should be gathered but says very little about how this should be done or what information is required for satisfactory traceability. Every DID (Data Item Description, documents to be written) contains a traceability section which defines what information elements it should be able to trace to, but the document itself has no clear definition of traceability.

IEEE/EIA (ISO/IEC) 12207[IEEE12207] specifies that traceability information should be gathered throughout the process, but says very little about what this information actually consists of. Further, the definitions used seem to vary with the context of the traceability information. It can be said, however, to be a good usage, since the standard also specifies that the entire process should be tailored to the task. An example from the standard: section 5.3.4.2, part of the definition of Supply Process, "The developer shall evaluate the software requirements considering the criteria listed below. The results of the evaluations shall be documented."

Matthias Jarke defines the following in [Jarke 1994]: “Requirements traceability, then, is defined as the ability to describe and follow the life of a requirement, in both a forward and backward direction, ideally through the whole system’s life cycle. Four kinds of traceability links are typically distinguished with respect to their process relationships to requirements:

- Forward from requirements. Responsibility for requirements achievement must be assigned to system components, such that accountability is established and the impact of requirements change can be evaluated.
- Backward to requirements. Compliance of the system with requirements must be verified, and gold plating (designs for which no requirements exist) must be avoided.
- Forward to requirements. Changes in stakeholder needs, as well as in technical assumptions, may require a radical reassessment of requirements relevance.
- Backward from requirements. The contribution structures underlying requirements are crucial in validating requirements, especially in highly political settings.”

In addition, Jarke talks about pre-traceability and post-traceability, the difference being the “timing” of the creation of traceability information. Pre-traceability is performed “in the process” while post-traceability is the creation of traceability matrices at the end of the project phase.

In [Scach 1996], Schach talks about traceability in the context of specification testing as follows: “It must be possible to trace every statement in the specification document back to a statement made by the client team during the requirements phase.” To aid in this process, requirements and specification documents should be cross-referenced and indexed. In the context of design testing, Schach defines traceability as the case where “every part of the design can be linked to a statement in the specification document”. The approach taken here is one in which traceability exists between more than just requirements and specification or requirements and code, but is relevant whenever any kinds of information can be linked together.

Deutsch and Willis define traceability in [Deutsch 1988] in the following terms: “Software is traceable when it is easy to find all the code that implements a particular requirement and, conversely, when it is easy to find all the requirements that are implemented by a particular portion of code. Traceability is important in verifying correctness and during maintenance.” This is clearly a requirements traceability point of view, defined in a context of software quality factors.

In DoD 2167A, Clause 4.2.8 (taken from [Gillies 1992]), traceability is defined as follows. “Traceability of requirements to design: The contractor shall develop traceability matrices to show the allocation of requirements from the system specification to the Computer Software Configuration Item, Top Level Computer Software Components, Lower Level Computer Software Components and Units from the Unit level back to the system specification. Traceability matrices should be documented in the Software

Requirements Specification, Software Top Level Design Document and the Software Detailed Design Document."

From [Sommerville 1995]: "Traceability between the requirements model for a program and the finished product is a means of assuring completeness. In addition, the traceability of parts of the program structure to the program's requirements model promotes simplicity. As a prime attribute of reliability, traceability also refers to the sufficiency and accuracy of the test cases used to determine how well the program satisfies the requirements set forth for it. In this sense, traceability is a quality technique. Generally, we can think of traceability in terms of a chain from the requirements model, through the development process, to the installation and final checkout of software."

2.4.4 Vertical Traceability

Traceability within a phase of the software development process is referred to as vertical, since it contains information related to different elements of the same phase. One example of this is to have requirements that lead to sub-requirements. The traceability information can then be used to link these together, so that if one of the main requirements changes, the effect on other requirements derived from it can be determined. Figure 2.5 shows how the relationship between top-level and lower-level requirements can be captured.

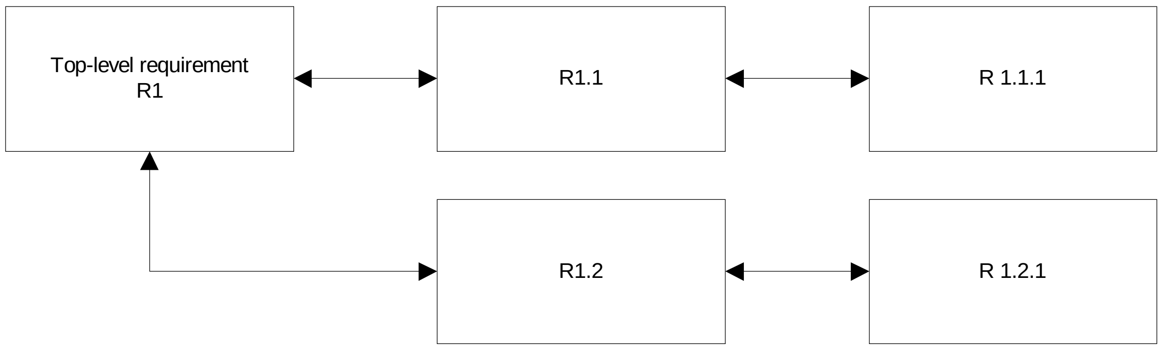


Figure 2.5: Flow-down of requirements

2.4.5 Captured information

A substantial amount of work has also been put into discovering what information makes sense in a traceability context. The main issue is how much information is needed for comprehension. The amount and detail of information gathered should ideally be customizable to the project group’s needs. Capturing too much information leads to information not being used and the task quickly loses importance to the people working with it. If not enough information is captured, the full context of the information might be lost and important knowledge goes to waste.

The “standard” approach to capturing traceability information is to capture relationships between information elements. These relationships can easily be modeled with hyperlinks, since either the relationship exists or it does not. The hyperlink method captures the existence of such relationships, but it does not say anything about the nature of the relationship.

One way of determining which information should be captured is described in [Gotel 1995] and is called “contribution structures”. The main goal of contribution structures is to capture not only what was done, but also who did it. Each artifact produced can have several people attached, all with different roles depending on their relationship to the artifact. Contribution structures enable project members to know who is responsible for a certain artifact, which can be helpful if the artifact needs clarification or refinement. However, there is a drawback this method. First, the tools needed to support such information gathering are complex, and place an extra workload on the user.

2.4.6 Summary of traceability work

As seen in the previous section, there are many definitions of traceability. While the difference between horizontal and vertical traceability is obvious, information about what is included within each type is not as clear. However, we feel that the IEEE definition [IEEEGST] is most widely accepted and so shall be used for purposes of this document. From this point, whenever we write traceability, the IEEE definition is implied.

2.5. Software development tools

2.5.1 Overview of software development tools

A thorough understanding of traceability tools and how they must integrate with tools currently in use, a look must first be taken at the types of tools normally involved in the development of software. The first tool that is usually involved is a text editor / word processor for writing the specifications documents, requirements, test cases and so on.

Only after all this is done is the software development environment used, where code can be written, compiled and tested. For this project, Microsoft Word has been used as the word processor, mainly because of its widespread use and ease of extension. For the software development environment, Visual Studio, another Microsoft product with good extensibility, was chosen.

For other projects, other tools might have been chosen, depending on the platform in use and the combined experience of the development team. However, the concepts described in this research can be extended to match other tools as long as they support hypertext and can be extended to match the descriptions contained in chapter three.

2.5.2 Overview of traceability tools

To compare and contrast state-of-the-art traceability tools with HyperTrace, several different solutions, most of them commercially available, were studied. In general, their greatest flaws relate to flexibility. This project's goal for a traceability toolkit is to have it be as general as possible and impose as few limitations as possible, although this does not seem to be a generally shared goal among software engineering toolkit builders Users who want to employ traceability techniques should be able to do so without switching the applications they are already using.

The most commonly used component seen in requirements engineering / traceability tools is a database. While this approach makes the storage problem for the links easy to solve, it also adds management overhead. Since the database requires configuration, users must be able to obtain correct access. One advantage of HyperTrace is that the links are embedded into documents, therefore ensuring correct user access.

It was decided at Hyper Trace’s inception that it would not be tied to any particular development model. While this is a common practice, it limits the usability of any software engineering tool. Tools that tie into a specific model are often inflexible, making the transition to and from other tools more difficult. HyperTrace offers a general solution that will allow any software team to employ and benefit from traceability.

2.5.3 Traceability tools

A number of traceability tools were studied in the course of this research and conclusions about them are offered in this section. A general overview of traceability tools can be found in [Gotel 1993]. The tools generally fall into one of two main categories, either horizontal or vertical traceability, although some are flexible enough to occupy both categories. There is little variation in information captured, which is mostly limited to “this information element is related to that information element” type linking, without description of the link.

The tools available go far towards providing traceability and requirements engineering. However, each of them has one or more of the following shortcomings:

- Lack of focus on requirements traceability. Many of the products are made for requirements engineering, which includes much more than just requirements traceability. In these situations, traceability tends to be de-emphasized. Some tools require that work must be accomplished in a specific manner, as defined by the company that built the tool. Such tools often come with other features as well. An example of this is Rational RequisistePro v4.5, which comes with a discussion board as well as the expected requirements engineering features.

- Lack of flexibility. Some of the tools directly mandate what can be added as traceability information. This lack of flexibility can limit the user's view of tool usability. This is particularly true when granularity is fixed. Some requirements are more complicated than others and might require traceability information to many more information elements than a simple requirement.

2.5.4 Rational RequisitePro 2000

Rational, best known for the Unified Modeling Language (UML), makes a requirements engineering product called RequisitePro. It ties in nicely with the other tools in Rational's collection of software engineering tools, but is much more than simply a traceability tool. Along with some of the other tools studied, it is more appropriately classified as a complete Requirements Engineering tool. While this can be a plus, it can also limit the interoperability with other tools that may be in use. Another aspect affecting tool size is complexity. RequisitePro requires a great deal of the user, in terms of training, knowledge and time. While additional complexity and a steeper user learning curve may be feasible for companies following very stringent rules in the software development process, they are not particularly compatible with traceability as such. Requirements traceability thus becomes just one of many features, not particularly emphasized and therefore diminished in value.

RequisitePro uses a database-solution tied into a Microsoft Word base. However, it forces the user to work with a specific set of operations, leading to very little flexibility. It also uses its own security features and access controls. While this makes any changes eminently traceable, it also adds to the administrative overhead of using the tool. Another part of RequisitePro management system allows users to write messages on a discussion board, where the progress of the project, details, clarifications and other interesting items can be discussed.

2.5.5 ARTS – Automated Requirements Traceability System [Dorfman 1984]

A classic requirements toolkit, ARTS was built by Lockheed Missiles & Space Company, Inc. Systems in the 1980s. Considering when it was built, it is a complicated database for requirements management. Using the parts borrowed from a NASA database project (RIMS), it consists of approximately 6200 lines of Fortran IV code. ARTS works by maintaining a vertical relationship between the requirements entered for impact analysis and verification of flow-down correctness. By maintaining a parent-child relationship from the top level down, requirements to the smallest details can be located, as can requirements in need of attention, even if top-level requirements change.

ARTS implements the necessary features for vertical traceability as defined in chapter two. It is interesting to note that the authors cited [Dorfman 1984] lack of traceability as a problem and said that computerized tools are needed to really benefit from the usage of traceability information. Sixteen years later, not much has changed, although tools are easier to come by. Also notable is the fact that in those 16 years not much has changed in terms of vertical traceability. While the tools have become graphical, the feature set is still comparable to that of ARTS.

2.5.6 SODOS – Software Documentation Support Environment [Horowitz 1986]

SODOS is an environment created for defining and manipulating documents related to software development projects. It uses a DBMS system for storing data and a window-based user interface. The documents to be edited using SODOS have to be predefined in the system. The SODOS environment supports the entire life cycle, from definition documents through specifications and code, all the way to the testing phase. Its main feature is that it supports relating document components. Users can then query this database in an SQL-like fashion later. Relationships can be either inter-document or intra-document, supporting a fine-grained relationship model. It is augmented with relationship types, like “derived from”, “derives”, “generates”, “implemented by”, “tests” and others (see [Gotel 1995] for related work on Contribution Structures). SODOS also supports automatic checking of document consistency.

SODOS is not a new tool, written in the early 1980s. However, it still stands as a very good implementation of horizontal traceability and continues to be referenced in many research papers.

2.5.7 The Evolution Support Environment System [Rama 1990]

The Evolution Support Environment (ESE) takes a different approach than most other traceability-enabled tools (like SODOS, and ARTS) in that it is based on version control. While able to locate versions of documents and code, it can also establish relationships between different information elements which can subsequently be used to establish a relationship between all the documents and files that make up a software project. However, this approach is not as intuitive as HyperTrace or any of the other applications studied here. One strong element of ESE is that it does support multiple versions with different relationships to related modules. This approach to traceability differs from the "normal" approach, but has merit in that it is very specifically directed towards code development and version control. For the prototype implementation, ESE developers used SCCS (version control software for Unix) and an INGRES relational database.

2.5.8 Summary of tools

It is clear that many attempts at building traceability and requirements management tools have been undertaken. However, it is also significant that creation of an environment not based on tools currently in use, produces too steep a learning curve to be of real value to developers. Developers, in fact, tend to be skeptical about learning a new tool, unless it offers huge benefits.

Another issue in the tool functionality debate revolves around the heavy reliance of some tools on databases, making the tool itself huge and requiring more software and hardware to function.

HyperTrace’s goal was to avoid both of these pitfalls, by creating a lean tool that integrates cleanly with the tools already in use by developers.

2.5.9 Summary of related work

Work related to this research project includes hypermedia research and software engineering toolkits, for both practical use and research.

CHAPTER THREE

3 HYPERTRACE ARCHITECTURE AND IMPLEMENTATION DESIGN

Chapter 1 of this document described traceability issue. In this chapter, a solution is described, along with an implementation prototype.

3.1. Introduction

As described in chapter 1, traceability is often not used in software engineering projects. Some of the reasons for this are lack of good tools and a lack of valuable uses for the information gathered. Traceability information that is simply placed in a matrix and does not receive any further use has limited direct value to the developer.

This project's proposed solution is to use hyperlinks for capturing traceability information. This solution is based on the concept that any software project can be viewed as a set of documents, with a set of editors that are able to manipulate different types of documents. The hyperlinks used to capture traceability information in this system must be able to link between these different editors, which can be accomplished in a generic way.

3.2. Traceability hyperlinks

3.2.1 Concept

The concept of using hyperlinks for traceability is an adaptation of the concepts used in hypertext discussion tools, like I-SHYS [Garg 87] and gIBIS [Conklin 1988]. A software project consists of numerous documents, including source code, specifications, and test cases, defect tracking documents and design documents. By inserting links into the documents that point to specific locations in other documents (or the same document), relations between and within those documents can be captured and subsequently easily traversed later.

To maintain a usable and simple system, the hyperlinks are considered unidirectional. Although bi-directional links have useful features, they can easily be simulated by using two unidirectional links. By making the links unidirectional, requirements on the insertion, modification and maintenance of the links can be relaxed. Insertion of links to applications outside of the HyperTrace system can also be supported as long as the application can open both types of links.

To use hyperlinks for traceability in this context, it is necessary to be able to insert links into these kinds of documents, regardless of the application used to edit them. Having these links inserted allows users to open files for which applications are not open, without having to start the application and open the file manually. This feature makes the information easier to access, since it is not necessary to know file locations or which applications to use.

A very similar approach (to gIBIS and I-SHYS) was chosen by Hermann Kaindl in [Kaindl 1993]. In RETH (Requirements Engineering Through Hypertext), they designed a hypertext system to support links for requirements engineering purposes. RETH is mainly designed for use in requirements engineering and does not support further software engineering tasks later in the lifecycle. By using hyperlinks, it is easy to model the relationship between different components in the system (i.e. showing where subcomponents belong). The hyperlinks allow for organizing requirements clearly (after where they belong in the system), which the author hoped would make it easier to create a complete set of requirements. However, RETH only existed in prototype form where link insertion was manual.

3.2.2 Practice

There are a couple of situations where hyperlinks can inserted into the different documents created in the process. Three of the ways in which traceability hyperlinks can be used are listed below.

During development, links between related information can be inserted as that information is created. By adding links while the work is being done, the added work overhead is minimal, since the user already has both documents open.

While developing, someone wondering where/how something is described can click on a link to go to the originating document. For example, a developer looking at a particular piece of code and trying to figure out how it should be tested could simply click on the link going to (one of) the test cases related to that code.

During code review, multiple developers can access the information necessary to verify that requirements are implemented correctly. By having instant access to any related information, the review process will be smoother and quicker. This feature comes from having all the related information just a couple of mouse-clicks away, even for people who are not familiar with the project.

3.3. Design goals

The HyperTrace tool has been designed with several advantageous features in mind:

- **Minimum administration overhead**

  The HyperTrace tool is designed to require minimal administration. After installing the system, the user should be able to use it without having to obtain permissions to access a database or other shared resources. All that is required is access to documents being worked on.

- **Minimum memory footprint**

  The memory footprint of HyperTrace should be minimal, which again comes from its promise to do one thing (manage the hyperlinks) and do it well. HyperTrace will not be inflated with unnecessary activities.

- **Ease of use**

  HyperTrace should be so easy to use that little or no training in direct use of the tool itself should be required. Some standardization within project teams on the usage of this tool may be required, but the tool itself should be simple and intuitive.

- **Non-intrusive to developers**

  The HyperTrace tool should not obstruct the development environment more than necessary or require developers to change the way they work.

- **Ease of integration with new tools**

  The HyperTrace architecture is designed to be easy to integrate in any type of application that could be used in a software development process.

- **Freedom to choose process.**

  HyperTrace is not tied to a particular development process. It should therefore be able to support traceability for any type of development process. This is compatible with the concept that developers should be able to use the tools they already know without interference. HyperTrace should enhance the traceability process while requiring little change in the way developers already work.

3.4. Architecture

In the following section, the generic architecture of HyperTrace is defined. HyperTrace-enabled application means any application that supports the use of hyperlinks as defined below and HyperTrace-enabled environment is a set of HyperTrace-enabled applications sufficient for software development.

3.4.1 Overview

The HyperTrace architecture is built around self-sufficient applications so that every application is capable of opening a hyperlink to a document can subsequently be edited.

Every application is also able to create links to any location in a document being edited. This level of isolation lends itself to easy integration and implementation of supported tools, given that the extent of the required functionality is so limited.

Every application that is to be integrated into a HyperTrace-enabled environment needs to be able to do the following:

1. Insert hyperlinks into its documents. This is required so that the system can embed links into the documents. The embedded links need to be as flexible as possible so that their targets are not constrained to specific types of applications.
2. Follow links to other documents, created by the same application or another HyperTrace-enabled application. This includes the ability to activate on a hyperlink, locate the required application to open it and initiate opening of the hyperlink.
3. Open a document and find the location pointed to by a hyperlink. The application needs to be able to find where opened links point. It also needs to have an interface with which other applications can communicate the link to be opened. This requires that the interface with which links are opened is generic, since the target is not known at design time, if the system is to be flexible and extensible.

An optional feature that is beneficial for HyperTrace-enabled applications is to be able to use the link exchange feature. This optional module enables two HyperTrace-enabled applications to exchange links automatically, instead of requiring the user to manually create links and copy to other documents. This is not a required module, since the applications can follow and insert links without it, but it further reduces the amount of work needed to insert links thereby reducing chances for errors in the link insertion process.

3.4.2 Block model of HyperTrace in action

To further illustrate the operation of HyperTrace, the basic operations that HyperTrace-enabled applications can perform (involving more than one application) are defined using block diagrams below:

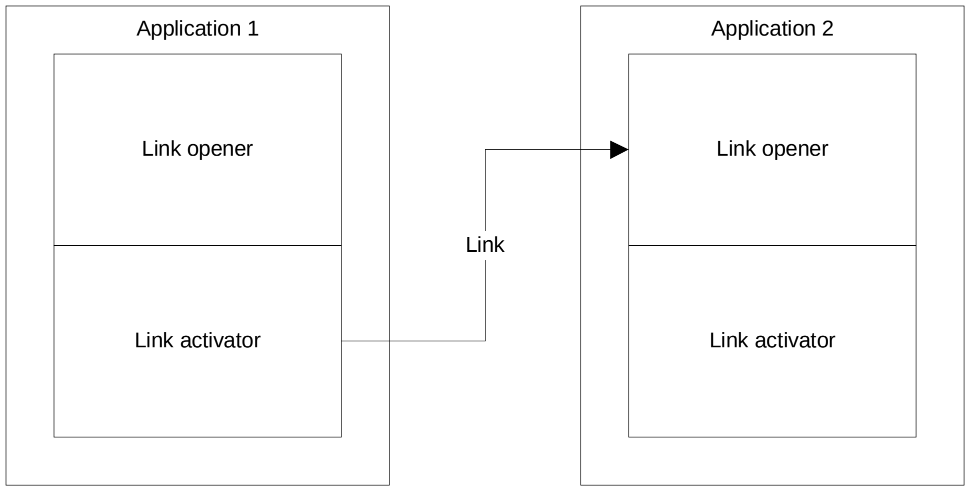


Figure 3.1: Link traversal

Figure 3.1 (above) shows how the two HyperTrace-enabled applications communicate when a user is traversing a link. The user clicks (or otherwise activates) a link in application 1. The link activator in that application then determines what the link is and what application is responsible for that type of link. It then contacts the link opener module of the other application and tells it what the link is. The second application will then open the document that the link points to and find the exact location that the link points to.

For link exchange, the situation becomes a bit more complicated, since there is bi-directional data exchange involved. This also requires an external mediator to be involved in the process. This makes the process more general and the application does not have to be concerned about which application it is exchanging links with. Figure 3.2 (below) shows how the applications communicate. It is assumed that application 1 is the first to communicate, but it doesn't matter which one of the applications are first, since the exchange protocol will take care of the necessary details.

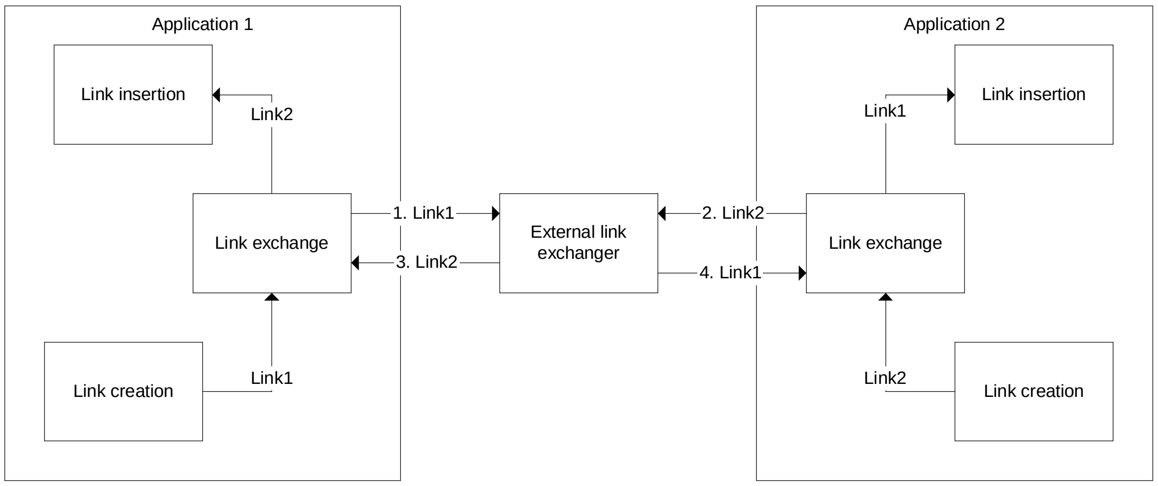


Figure 3.2: Link exchange in operation

As shown in figure 3.2, the first thing that happens is that application 1 creates a link in its document (link1). This link is then passed to the external link exchanger, which handles the transaction. Since there is no link available for application 1 yet, it will wait until one becomes available. When application 2 enters the transaction, it creates a link to its document according to the user's commands (link2). This link is then passed to the external link exchanger. When this happens, this link (link2) is passed to application 1, which has requested it. Link2 is passed to the link exchange module in application 1, which passes it to the link insertion module where it is inserted into the document currently opened in application 1. Application 2 then receives the link (link1) from application 1 and passes it to its link insertion module for insertion into the document. This completes the transaction.

Clearly, this generic way of exchanging links is superior in flexibility to a solution that requires the application to know which other application it is going to exchange links with. This is a seamless operation, which contributes to the ease of using this system.

3.4.3 Differences with previous approaches

When compared to the traceability and requirements engineering tools described in chapter 2, the HyperTrace system has a couple of distinct features.

- **Storage.** The HyperTrace system uses no external storage for storing links or other meta-data. This is different from other concepts, such as the Open Hypermedia Protocol, where it is thought that all links should be stored in a central database. By not requiring a database to store anything in, administrative overhead and the memory footprint of the system are reduced as ease of use is increased. DHT [Noll 1994] performs somewhat similarly, but the system uses servers to store all the documents and links between them.
- **Integration into applications.** While some of the traceability toolkits use a single application for all work related to the project, like SODOS, HyperTrace recognizes the fact that a single application for all users is not going to work.

HyperTrace integrates cleanly into the host applications, leaving the users with tools they already know, while adding functionality.

- **Flexibility.** The HyperTrace system is designed to be able to support any type of linking, not limiting itself to just doing one type of links. Some of the traceability and requirements engineering toolkits on the market are strictly designed to one thing, one way. One example of this is Rational RequisitePro, which is a good tool for requirements engineering. In order to obtain full benefits from the tool, the user must follow a designated process based on UML, not only for the design, but also clear through development.

3.4.4 System strategy

The link strategy employed in HyperTrace is based on HTML-style links, which can be embedded in most applications used for software development. By using hyperlinks, we can easily hide details such as which applications are used and where the files are stored, as long as the system can locate the applications and files. This can all be accomplished in a way that is transparent to the user, which is a strong benefit of hyperlinks.

Hyperlinks are also often used in other applications for cross-referencing (which is very much like traceability) and capturing related information. This matches the way hyperlinks are used in this system and makes it compatible with many of the same semantics that are used elsewhere. One very important benefit derived from using the same semantics is increased user familiarity. Most users of a software development system are accustomed to using the World Wide Web (WWW) and by mimicking this way of linking traceability can be made more intuitive.

3.5. System architecture model

3.5.1 HyperTrace tool

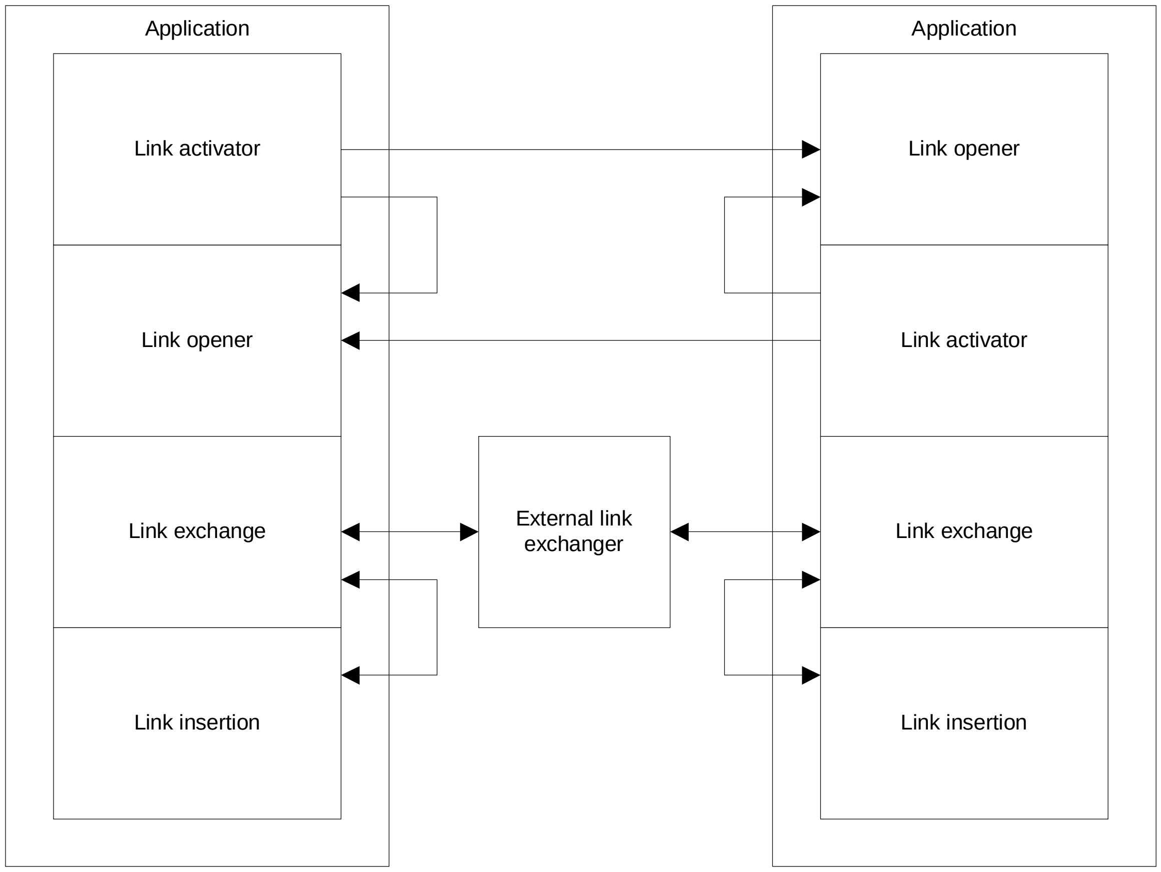


Figure 3.3: System architecture

Figure 3.3 shows the system architecture with the different modules in the HyperTrace tool and which modules they communicate with. This not only outlines the different components, but also clarifies which interfaces need to be implemented in which module.

The link activator is the module that is activated when the user clicks (or otherwise activates) on a link in the application. It will then locate the link opener module in the application (note loop-back if link points to a document that is opened by the same application as the one in which the link activator is running) that is required to open the document from the link. If the target application is not running, it will be started. The link opener will then open the document and find the location in the document that the user requested.

The link exchange module communicates with the link insertion module in the same application and with the external link exchanger. As shown in figure 3.2, this is done since the link insertion module is responsible for creating links in the document and inserting links to other documents in the open document.

Since the modules are self-contained, the implementation of these can be accomplished in several different ways. This helps reduce the complexity of the implementation and reduce chances for defects in the implementation (simpler code is easier to get right).

3.5.2 Link insertion module

This is one of the two modules that actually are invoked by the user directly. It is responsible for inserting links into the current document. This requires the ability to insert links pointing to other documents as well as inserting links that other applications can use to point to locations in documents opened by the application.

The process for inserting links in the currently opened document is shown as follows:

1. Create an identifier for the current location, containing the information required to find this location later. This usually includes a full path to the file plus an identifier to the location in the document.
2. Insert (if necessary) a tag so that the location can be found later. This can be hidden from view (ideally) or visible to the user, depending on what the applications support. This is used to find this location later.
3. (Optional, required if item 4 is not implemented). Insert the link into the document so that it can be copied and inserted manually in other applications.
4. (Optional, required if item 3 is not implemented). Communicate with the link exchange module to exchange the link with some other application.
5. (If step 4 is included). Receive a link from the link exchange module and insert this into the document.

Step 3 can be omitted if step 4 is implemented (and vice versa), but one of the two needs to be included so that the link can be inserted into other documents.

3.5.3 Link follower module

The link follower module is responsible for opening links in the application the user activated. To do this, the application will have to go through a series of steps as outlined below:

1. Locate and parse the link the user clicked on. This is dependant on how links are embedded in the document (by the link insertion module).

2. Locate the application responsible for opening that type of link
3. Communicate with the other application.
    a. If the application is running, invoke the link opener module on that application and inform it of the link that it should open.
    b. If the application is not running, locate the application and start it, before invoking the link opener module on that application.

The process of locating which application is required for which type of link is not specified, since that is highly implementation specific. For example, this could be done in the same way the operating system locates applications to open files that are opened in the file manager. Other ways might include using a centralized database of link/application mappings that might be more feasible, depending on what support the operating system has.

3.5.4 Link exchanger module

The link exchanger module is responsible for taking a link from the link insertion module and passing it on to the external link exchanger. In return, it receives a link to a document in the other application involved in the transaction. This received link is passed back to the link insertion module where it is inserted into the document. This is completed using transactions, where each transaction involves two applications and one link exchanger. See figure 3.2 for a more complete description of this transaction.

An invocation of the link exchanger contains the following steps:

1. The link exchanger is invoked by the local link insertion module, which has created a link that is going to be exchanged with some other application
2. The link exchanger invokes the external link exchanger and sends the link over.
    a. If the link exchanger has a link ready, the link exchanger sends this back to the application.
    b. If the link exchanger does not have a link ready, it waits for one to become available (possible timeout).
3. Pass the received link to the link insertion module

3.5.5 Link opener module

The link opener module receives links from a link follower module, either in the same application or from some other application. It opens this link in the application after locating the required documents. The steps required to do this are:

1. Receive the link
2. Parse the link to find out what document and what location in that document is requested.
3. Open the requested document
4. Find the requested location within the document.

3.6. Implementation

The HyperTrace concept has been implemented in a limited prototype as part of this research. The following sections contain a description of this prototype.

3.6.1 Implementation prototype

The prototype is implemented as an extension to Microsoft Word and Microsoft Visual C++. We chose to implement the HyperTrace prototype using well-known tools that are extensible, making use of a great deal of functionality that already exists within the tools.

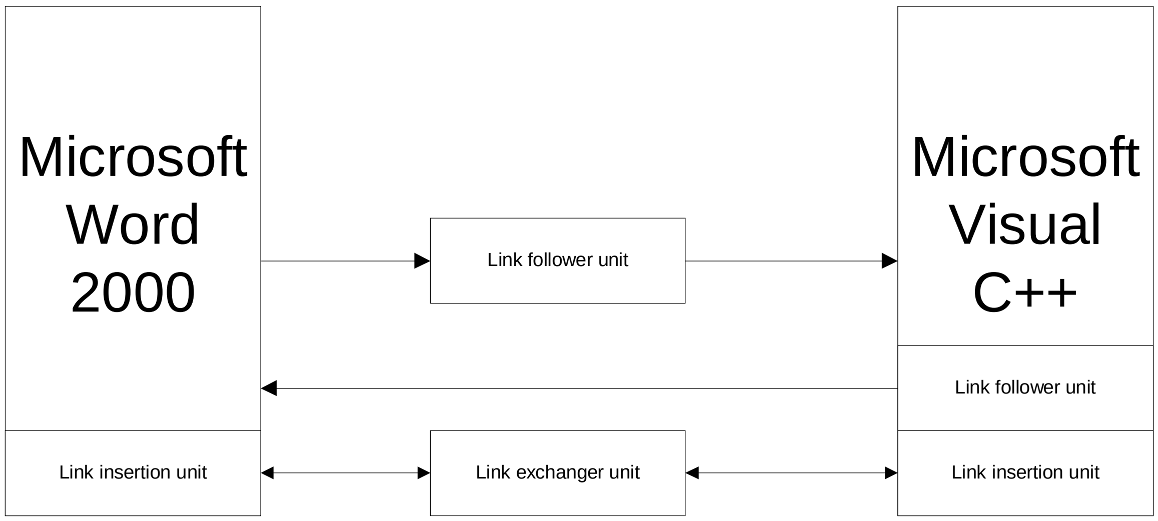


Figure 3.4: HyperTrace implementation prototype architecture

Figure 3.4 shows Microsoft Word and Visual C++ with the modules used to implement the necessary functionality to turn them into HyperTrace-enabled applications.

3.6.2 Extension architecture

To extend Microsoft applications to include the required functionality, it was necessary to use the predefined extension architectures present in these applications. The Microsoft Office suite of applications uses Visual Basic for Applications as their macro language, enabling customization work through scripting. For Visual C++, extensions can be written using either C++ or Visual Basic scripts, depending on the functionality required. These technologies are described below to give an idea of what opportunities for extension exists within these applications.

Microsoft's COM architecture is also described, since it was used to create the external link exchanger for the prototype.

3.6.3 Visual Basic for Applications

Visual Basic for Applications (VBA) is the macro programming language for the Microsoft Office platform and is built on the concepts of Visual Basic. One of the main strengths of VBA is that it can access properties and methods of the host application to customize its behavior By doing this, advanced features can be built with few lines of code and leverage the fact that the Microsoft Office platform has a lot of functionality already included within it.

An example of this is the way the Link Exchanger module is written. Using about 70 lines of simple VBA code, it can look up a COM object by name, invoke methods on the COM object and then use the returned text to insert a bookmark and hyperlink into the Word document. The use of internal Word objects and methods is the same as the functionality that can be built from an outside application using COM Automation, having most major objects in Word exposed to outside invocation.

3.6.4 Visual C++ add-ins

Using the Visual C++ add-in interface allows extension of the Visual C++ work environment. The add-ins can be written in either Visual C++ or in VBScript, implementing functionality not present in the environment.

The add-ins are registered with the development environment and are then loaded upon starting the environment. An add-in adds commands to the environment, commands that can be activated either through COM Automation or by adding to the Visual C++ toolbar. The add-in interface is a class definition that is inherited and customized to fit the needs of the add-in. The interface includes methods for registering the component upon installation and one for initializing the component upon starting the environment.

From within the add-in components, the Visual C++ environment can be controlled as it can from the outside, using COM Automation. The classes exposed through that interface range from the application (main control of the environment) down to the text editor windows. Functionality can also be added by controlling the build process for projects.

As with the VBA macros for the Office environment, having the ability to extend the environment from the inside really adds opportunities for extending the tools in ways not previously possible. It allows for a tighter integration than would be possible for something that would have to live on the outside of the environment since you can actually manipulate the environment and their predefined functionality.

3.6.5 COM – Component Object Model

Microsoft’s COM architecture consists of several technologies merged into one name. Although based on the same architecture, they have differences that reflect on their usage. [Rogerson 1997] gives an overview of what COM is and what it is not. In essence, COM is a specification of how components can be designed to interact based on interface inheritance (in object oriented environments). COM allows for clients to access servers even when the clients are not written in the same programming language as the server.

Some of the technologies based on COM are COM Automation (the ability to externally call methods on objects residing within other applications), DCOM, COM+ and most of the ActiveX.

COM has been in the works for a long time and can be traced back to the early versions of Microsoft Windows and how objects can interact in that environment (everything from drag and drop to the clipboard). This was then extended and formed the basis for the OLE (Object Linking and Embedding). OLE made it possible to embed documents or parts of documents into other documents across applications and then later editing them using the application that understands that document type.

In the later years, COM has been extended through technologies such as Distributed COM (DCOM) to compete with CORBA in the field of distributed objects. DCOM supports method invocations across machines, but is otherwise the same as COM.

It has also evolved into COM+ [Platt 1999] which supplies many services to COM. COM+ was introduced alongside Windows 2000 and is set to make the development of enterprise level applications easier.

3.6.6 Link formats

The links are embedded into documents in ways customized to the application. Since this is application specific (creating a generic link embedding would require the applications themselves to be developed with the linking in mind), what these links look like also varies from application to application.

3.6.6.1 Microsoft Word

For Word, embedding links is easy. The tag required to find a link is a Word bookmark, which is invisible to the user and can be created on the fly without modifying any functionality or the document structure.

For this prototype, it was required that bookmark identifiers be automatically generated. To do this, the current time is used to generate the identifier, since we can be sure that no two people will work on the same document at the same time inserting a link, since the granularity chosen is one second By calculating an approximation to “number of seconds passed since January 1. 1970”, we have a number that is monotonic growing for all time. This assures that there are no two identical links in the same document. However, Word does not allow bookmarks to start with a digit, so the number is appended onto a 1-character string.

This caused link to a location within a Microsoft Word file to receive the following format: document_name#bookmark, for example:

```
“C:\demo\doc\ToolSpecification.doc#l954879452”.
```

Figure 3.5: Format for link to a location within a Word document

Outbound link in Microsoft Word are inserted as hyperlinks, which are also supported. It was decided that the prototype should use as much of the pre-existing functionality as possible. Another benefit of this is that Word will automatically underline the links and use a different color to display them, so they are easily locatable.

3.6.6.2 Visual C++

Inserting links into Visual C++ required an approach that is a bit more thorough. Since the current location is more than just a file and a location, the current project is also required. This is more consistent with the way Visual C++ works and opening the project is required to open a file.

In the same way as for Microsoft Word, it was decided that time is a good identifier to embed in order to find a location in the document. Visual C++ allows the extending component to directly access the clock, so finding the number of seconds passed since January 1. 1970 is easy. This is then embedded in a comment using the format shown in figure 3.6.

```
// link=961019447
```

Figure 3.6: Link format for embedding into Visual C++ source code comments

This results in the complete link looking rather complex. Since Visual C++ does not have sufficient link opening functionality, we had to use an external program to open the documents. This is more described in the section about the link follower tool and the components required for it, but for completeness, it should be noted that it includes a program that listens for TCP connections on port 90, the same way a web server would listen to port 80.

This was done so that links to Visual C++ source code used the same conventions as normal web links, which allowed use of the hyper-linking code to be implemented as part of Microsoft Word to open the hyperlinks.

The complete link that is inserted into Visual C++ is shown in figure 3.7.

```
//http://localhost:90/cgi?project=C:\\code\\BSTRExchange\\BSTRExchange.dsp&filename=S
```

Figure 3.7: Link format for Visual C++

As figure 3.7 shows, the link is formatted like Internet URLs (Uniform Resource Locators). The first part (http://localhost:90) describes that the server that should be contacted to open this link resides on localhost, listening to port 90. The /cgi? part is required for compliance with URL formatting. The rest of the link consists of project=<absolute path to project file>, filename=<filename within project> and target=<id>, where id is the same as shown in figure 3.4.

Since Visual C++ does not support hyperlinks in the same way Microsoft Word does, we needed a format to describe outbound links. To support this, the choice was made to embed using the format shown in figure 3.8.

```
// $doc: <link>
```

Figure 3.8: Outbound link format for Visual C++

This allows the link follower module to parse the links correctly.

3.6.7 Link Insertion tool

The link insertion tool implements the requirements for the link insertion and link exchange modules within the applications (as defined in Figure 3.1).

3.6.7.1 Microsoft Word

For Microsoft Word, the link insertion tool consists of two macros, InsertLink and InsertBiLink. The InsertLink macro inserts a bookmark in the current document, and produces the link that points to the newly inserted bookmark. This link can then be copied as normal Word formatted text to other documents while still being usable as a link.

The InsertBiLink macro is similar to InsertLink, just extended to exchange links with Visual C++ (or other tool capable of exchanging links). It generates the bookmark and link in Word and then sends this string to the External Link Exchanger unit, which will receive a link from the other application involved in the exchange and then return this to

Word. This link is also inserted into the Word document as the outbound link for that piece of information. Figure 3.9 shows the Word toolbar with the InsertBiLink and InsertLink macro buttons on the second row of buttons.

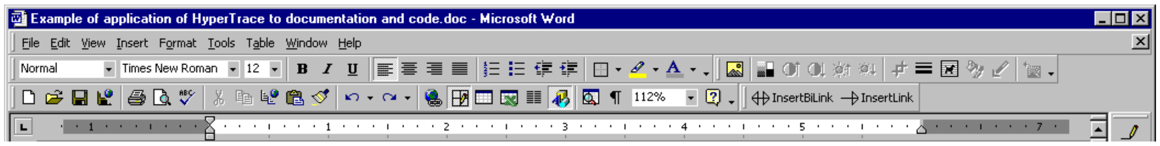

Example of application of HyperTrace to documentation and code.doc - Microsoft Word
File Edit View Insert Format Tools Table Window Help
Normal
Times New Roman
12
112%
InsertBiLink
InsertLink

Figure 3.9: Word toolbar

3.6.7.2 Visual C++

The link insertion unit in Visual C++ consists of two methods, InsertLink and InsertBiLink. The InsertLink method inserts a link "target" in the current document and produces the link that points to the newly inserted target. This link can then be copied as normal ASCII text to other documents while still being usable as a link. Since it contains http://, most programs in the office suite will recognize it as a link and insert the necessary information around the text. The unit also inserts a tag in the source code, which is necessary to find this location again. This is a comment of the form "// link=<id>" where <id> is the same as the <target> put in the target=<target> part of the link.

The InsertBiLink method is similar to InsertLink, just extended to exchange links with Microsoft Word (or other tool capable of exchanging links). It generates the bookmark and link in Visual C++ and then sends this string to the Link Exchanger unit, which will receive a link from the other application involved in the exchange and then return this to

Visual C++. This link is also inserted into the Visual C++ document as the outbound link for that piece of information. Figure 3.10 shows the Microsoft Visual C++ toolbar with the two added arrows on the right hand side (first row of buttons).

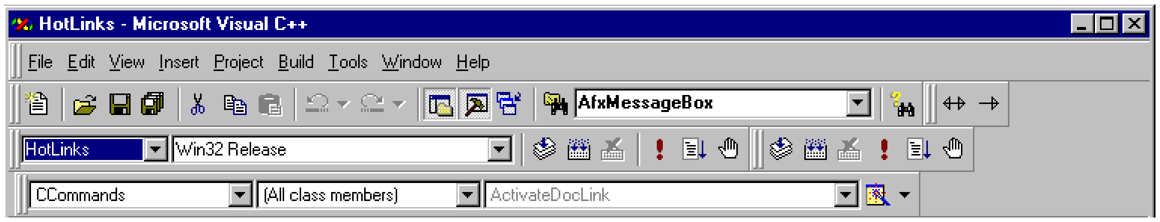

HotLinks - Microsoft Visual C++
File Edit View Insert Project Build Tools Window Help
AfxMessageBox
HotLinks
Win32 Release
CCommands
(All class members)
ActivateDocLink

Figure 3.10: Visual C++ toolbar

3.6.8 Link follower tool

By using hyperlinks in the same way they are used in such applications as the World Wide Web and Microsoft Office, we can benefit from the already existing code to follow hyperlinks. Therefore, the task of implementing the Link Follower code in Word has already been completed by Microsoft, making our job easier. However, Visual C++ does not have a built-in way of hyper-linking to source code. Figure 3.11 shows the structural overview of the Link Follower module, with the Link Follower having code inside of Visual C++ and in a standalone process.

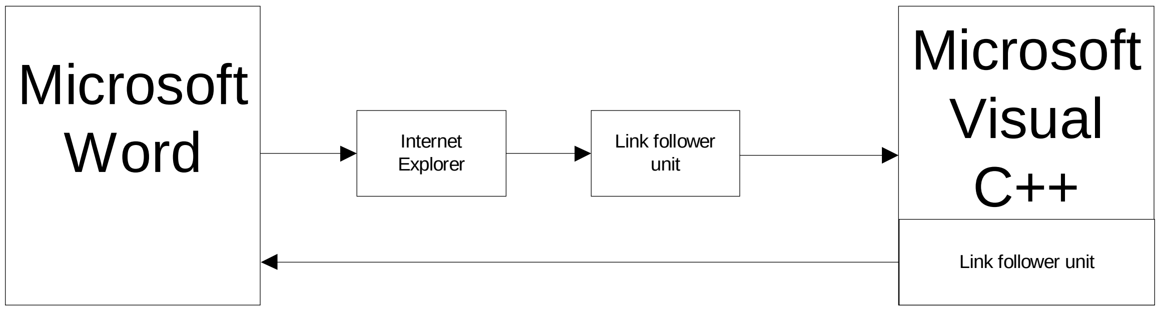


Figure 3.11: Overview of the link follower module

3.6.8.1 In Microsoft Word

As previously stated, the code for following hyperlinks is already implemented in Microsoft Word and other applications; therefore all we have done is to have the link insertion module embed hyperlinks that these applications understand into the document. The traversal of links is already implemented in the application.

3.6.8.2 Microsoft Visual C++

Links pointing to locations in Visual C++ source code are of the format previously described, http://localhost:90/cgi?project=<project>&file=<file>&target=<target> where project is the project file that should be opened (absolute path required), file is the name of the file that contains the target (relative path is sufficient) and target is the location requested. However, since Visual C++ does not have any code built in to receive such links, we have built an http-like server that listens on a different TCP port (port 90). This gives us the localhost:90 part of the link. This http-like server receives http requests and parses the project file name, file name, and target. It then uses COM Automation to control Visual C++, which opens the project, opens the file, and then finds the target.

Finding the target is accomplished with a simple “search the file” for “//link=<id>”. One advantage of using the “// link=<id>” is that it is not based on the line number, allowing changes in the code to happen without disturbing the link, as long as the line containing the link is not modified.

The link listener could have been registered with Windows as a handler for some protocol instead of using http, but such implementation details don’t change the semantics of the operation. However, it would remove Internet Explorer (or the program registered for http:// links) from the path to the Visual C++ code.

Following hyperlinks from a Visual C++ source code file also requires additional code. The plug-in detects double clicks on text, checks if it contains the “// $doc:<link>”, and opens the link if it does. The code that is invoked upon parsing the link and deciding which program to use is generalized to the point of checking the extension of the file.

File extension	Program invoked
Html, htm	Internet Explorer
Doc	Microsoft Word
Unknown	Windows default for extension

Table 3.1: Supported file types

Since the decision is made based on file types, this can easily be extended to support more file types, as long as the application needed to open the file supports some form of automation interface. Table 3.1 shows the extensions of documents currently supported.

3.6.9 External Link Exchanger

The External Link Exchanger unit was constructed to facilitate insertion of bi-directional links. It is implemented in a COM (Component Object Model) object that will be automatically loaded by Windows when needed. It stores two strings, one for each application in the transaction. The first application will call putString1 (string) and the object will store this string for when the second application calls getString1, fetching that string from the COM object. The same goes for the second string, however it will be inserted by the second application in the transaction. Figure 3.12 shows the structure of the transaction. Note that both InsertBiLink methods/macros are involved in this transaction, making the Link Exchanger unit a synchronizing unit for the transaction.

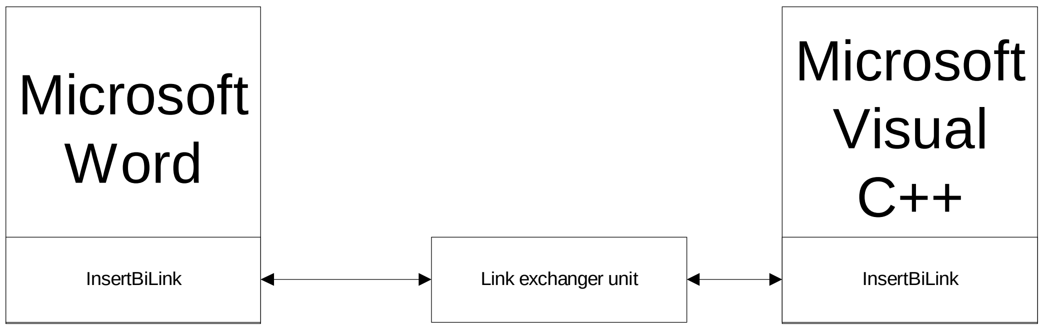


Figure 3.12: Exchange of links

By using a COM object for the Link Exchanger unit, any Windows program can be extended to exchange data with it, since it occupies its own separate process space. By having the UID (Unique Identifier) of the COM object, it can be located and from there, one can invoke methods using any programming language.

3.6.10 Implementation status

All the main pieces of the HyperTrace prototype have been implemented. This includes the Link Follower modules and Link Insertion modules for Visual C++ and Word. The Visual C++ link follower module is restricted to HTML files residing on remote servers and Word documents. The HTML files stored locally cannot be used, since Internet Explorer does not recognize the anchor part of the URL if the file is stored locally. However, the amount of code required to extend this functionality to new applications is minimal, since the Visual C++ uses a set of if-then-else statements based on the file type to traverse links.

3.6.11 Microsoft Windows 98

The current implementation does not run under Windows 98. While this was tested with the same applications (Microsoft Word and Visual C++), it did not work. The traversal of links from Microsoft Word to Visual C++ is broken, with Visual C++ not being able to open the project specified in the link.

3.6.12 Implementation issues for integrating new applications

Integrating HyperTrace into Word posed little or no difficulties. This was not the case for the integration into Visual C++, which consumed most of the time used to implement HyperTrace.

3.6.13 Integration in Visual C++

Both the link follower and the link insertion modules are built as Visual C++ plug-ins. This is an attempt from Microsoft to allow third party extension modules to be built. However, the work required to build these is not trivial. One obstacle is the testing and debugging of such plug-ins. If the code in the plug-in crashes for whatever reason, Visual C++ (the IDE) dies with it, leaving no trace of what happened. In addition, debugging is harder, since the plug-in is residing within the Visual C++ IDE and a second Visual C++ debugger is required.

The most obvious obstacle, however, is with the link follower module. When closing the Visual C++ IDE, it causes a memory access violation in Visual C++ that causes an unclean exit from the environment. This has been verified to be in termination code in the plug-in, but a fix has not been found. It has, however, been checked that this does not in any way harm the Visual C++ environment since all the files the user was working on have been saved and closed prior to the access violation.

One thing that has not been implemented for the Visual C++ environment, but would have been helpful, is text highlighting. Having the hyperlinks be colored or formatted for higher visibility could really help the readability of the code and links, making the tool easier to use. Hopefully, with some work, this can be done either through cooperation with Microsoft or working around the system and modifying the text highlighting code to take “$doc:” into account when formatting comments.

3.7. Link Exchanger module

The link exchanger module is a single-threaded COM object. The choice of single-threaded instead of multithreaded is one of simplicity of method invocation and mutual exclusion. However, this was found to limit the implementation of the clients that connect to this object. Since there can only be one method invocation active at any given time, the client cannot wait for the string to appear, but has to regularly check (poll) the object to see if the value is available. Consider this short example: application 1 sends its string to the link exchanger (putString1). After successful invocation, it calls getString2, which returns error since the second application has not yet called putString2. Application 1 is forced to continuously call getString2 until it succeeds. If the method getString2 was to wait until putString2 was called, the object would be blocked and putString2 could never be invoked. One work-around for this is to use polling, which is how the clients were ultimately implemented, but this solution is not as elegant as it could have been if multiple threads had been used.

3.8. Integration issues to be expected in other tools

If the two applications studied in this project are typical, and they are expected to be, the issues facing implementation into further applications depend heavily on the type of application. This goes back to the work done by Dr. Whitehead in [Whitehead 1997] in classifying the integration styles. Using wrappers makes it easier, since the tool already supports some or all of the required functionality. Tools with hypertext functionality, like Word, will integrate with HyperTrace without much difficulty. However, an application where hypertext functionality is not available requires the developer to come up with a way to embed hyperlinks in the document and traverse hyperlinks into a document. The way this was done in Visual C++ for this project can serve as a model on how this can be done, but in the end the customization is defined by what the application supports and what it does not.

3.9. Possible improvements to resolve integration issues

To reduce the integration issues for applications, several factors must be taken into account. Clearly, having hypertext functionality in the application helps, since this is where most of the complicated code lies. Although not studied as a part of this research, the integration with Open Source applications holds some promise. The idea of having the source code to the application to be integrated is valuable, since it can then be changed in every direction needed for the integration effort. This is not automatically beneficial, however, since understanding the code and where to change it can be time consuming.

3.10. Problems and issues currently in prototype

The prototype as it stands has a couple of issues that are not caused by the prototype itself, but rather by limitations in the environment into which it has been built. The major one of these is that unless the user is logged in with administrative privileges, he or she cannot load any add-ins to Visual C++. This is a severe limitation that Visual C++ has and cannot easily be worked around unless the restrictions on registering dynamic linked libraries (.dlls) in the system are lifted.

Other issues current in the prototype is the design of the link exchanger. Since COM in a single-threaded mode does not allow two applications to call method in one object at the same time, the calls need to return failure until the other application has put the link into the object. This could be solved by going to a multi-threaded object, but this does not easily lend itself to implementation.

3.11. Summary of prototype

The prototype has been implemented and confirmed that the design goals were, in fact, reachable. The concept has been proven to work and in chapter 4 the prototype will be demonstrated on projects.

3.11.1 Summary of architecture and implementation

The design in Figure 3.1 has been described and fielded as a possible solution to the problem described in the first chapter. We have shown that a prototype implementation of the system has been constructed.

CHAPTER FOUR

4 HYPERTRACE IN USE

4.1. Introduction

This chapter describes how HyperTrace operates within the applications with which it is working. With the exception of link insertion, which is application specific, HyperTrace functions much like other hypertext tools do in most applications for similar purposes. This section explains the insertion of links (unidirectional and bi-directional) as well as traversal of links, both from a user's perspective. Two examples are provided that further explain the benefits of using HyperTrace. Some possible metrics are discussed to further expand the ways in which HyperTrace can be utilized. These relate to measuring the number of links inserted and how the properties of the project might be determined by looking at the links inserted.

4.2. Inserting unidirectional links

From Word, the user would simply click a button on the toolbar (or use a user-defined key sequence) to invoke the InsertLink macro. Subsequently, a bookmark would be inserted at the current location in the active document along with the text necessary to link to this location.

Figure 4.1 is an example taken from the HyperTrace specifications document:

Link to this location: C:\demo\doc\Tool Specification.doc#l954879452

Figure 4.1: Example of link to Word document

Performing the same operation in Visual Studio has much of the same semantic effect, although the text looks a bit different. Again, there is a toolbar button to click on, and a user-defined key combination, which can also be used. However, user-defined key combinations in the Visual C++ environment require a minor source code modification to the Link Insertion plug-in.

Figure 4.2 is an example taken from the Link Exchange project in Visual C++:

```
// link=961019447
//http://localhost:90/cgi?project=C:\\code\\BSTRExchange\\BSTRExchange.dsp&filename=StringExch
```

Figure 4.2: Example of link to Visual C++ source code location

As evident in figure 4.2, the first line is the link destination marker, which is used to find the location later. The second line is the URL of the link, which can be copied into other documents and used as a hyperlink there. Notice that the filename part of the link is relative to the project path, which shortens the link significantly, but that the project needs an absolute filename (the .dsp file). This is the minimum amount of information needed to open the Visual C++ project and source code file and find the location in question.

4.3. Inserting bi-directional links

When inserting bi-directional links, HyperTrace will first insert the link as specified above and then insert the link to the other document in the transaction. This bi-directional link insertion is done by clicking on the ↔ arrow on the toolbar in both applications.

```
Link to this location: C:\demo\doc\Tool Specification.doc#l955115035

Link:http://localhost:90/cgi?project=C:\\demo\\code\\BSTRExchange\\BSTRExchange.dsp&filename=StringExchange.cpp&target=96
6285932
```

Figure 4.3: Example of bi-directional link in Word

Figure 4.3 shows the result of the transaction in Word, while Figure 4.4 shows the same for the Visual C++ source code.

```
// link=966285837
//http://localhost:90/cgi?project=C:\\BSTRExchange.dsp&filename=StringExchange.cpp&target=966285837
// $doc:C:\demo\doc\Tool Specification.doc#l955115035
```

Figure 4.4: Example of bi-directional link in Visual C++

4.4. Traversing links

Traversing links is accomplished as might be expected. In Word, clicking on the link performs the action the same way it does with hyperlinks to web documents. For Visual C++, double-clicking on the line of text starts the traversal routine.

4.5. Evaluating document

After inserting links in documents and source code during development, it can be beneficial to examine a few basic statistics about these links. Note that very few, if any, of these can be used to prove anything, but they can be used to measure properties of the project. These can then lead to further investigation to see if there is any cause for change. Listed below are some of the possible indications that one might hope to see (or not see) after obtaining the statistics.

4.5.1 Completeness

A requirements specifications document that does not have every paragraph (requirement) linked to source code might be incomplete. The fact that the link does not exist does not mean that the source code does not exist, but it can be an indication that further checking needs to be done on that requirement. The same thing applies to source code. A function or object that does not link to either design or requirements documents is potentially an object defined by the programmer with no basis in the software engineering process. On the other hand, the link might simply be missing.

4.5.2 Module complexity

A very complex module might contain many links relative to lines of code, indicating that either the module was over-specified or that the module is too big/doing too much. It is not clear at this point, what the numbers would look like since the metric has not been applied in any real situation with prior knowledge of complexity, but it can be an accurate indication.

4.5.3 Module interconnection

Depending on the organization of the documents, a given module should have links to certain parts of the document and not randomly scattered ones. If a module contains links to every part of the specifications document, or is overlapping with other modules, this can be an indication that either the specifications are poorly organized or the module is not following the specifications very well.

4.6. Example 1: HyperTrace

4.6.1 Introduction

The specifications document for HyperTrace was used as the first example (and testing platform) for the HyperTrace environment. However, it is a limited specifications document, since the requirements are few. However, it was found that the links were fairly concise and descriptive even for a small-scale specification. From the specifications document, there were 2 links for the insertion requirements (insertion and bi-directional insertion), one for each going to the Visual C++ module implementing the functionality and one to the Word macro that implements it. The problem with using VBA as the implementation language is that links could not be inserted into the macros. Requirements for the link exchanger module go to three different pieces of code, which is natural since the functionality is needed in three places (Visual C++, Word and the COM object).

However, since the specifications document and resulting source code are small, another example was chosen to really demonstrate the value of HyperTrace.

4.7. Example 2: JPEG-2000

4.7.1 Introduction

The JPEG 2000 standard (final committee draft) was recently used in a software engineering project at WSU. A group of students took the specifications provided in

ISO/IEC FCD15444-1 [JPEG2000] and turned it into source code, implementing the encoder and decoder.

The JPEG-2000 standard is intended to create a new image coding system for different types of still images with varying characteristics. This coding system is intended to provide low bit-rate operation with rate-distortion and subjective image quality performance superior to existing standards, without sacrificing performance at other points in the rate-distortion spectrum.

Because the project has a comprehensive specifications document with several versions of specifications, it serves as a good example of the benefits obtained by using HyperTrace.

However, the insertion of links into the specifications and code were not done while the project was in development. This means that the links were inserted afterwards by looking at the code and specifications to figure out what elements are related. Even with this obstacle, inserting the links was a straightforward task, confirming the goal of creating a process that is both easy and relatively quick.

4.7.2 Statistics and observations

The JPEG 2000 specifications document consists of 12 annexes (A-K). The source code is split into three different projects (encoder, decoder and wavelet transform). The links are distributed as follows:

Annex	Encoder source code	Decoder Source code	Wavelet transform
A	0	0	24
B	0	0	0
C	12	0	0
D	11	12	0
E	0	1	0
F	0	0	17
G	0	0	0
H	0	0	0
I	0	0	0
J	0	0	29
K	0	0	0

Table 4.1 : Distribution of links in JPEG-2000

That several annexes lack links from the source code is not surprising, given the document structure of the JPEG 2000 specification. Annex I contains the file format, H contains a description of Region Of Interest (ROI) coding, limiting itself to what is stored in the jp2 file.

Module	Links	Approx LOC	Author
Jpeg_encode	23	2200	A
Jpeg_decode	13	3100	B
Wavelet	66	2500	C

Table 4.2: Statistics for the JPEG-2000 source code

The statistics shown in table 4.2 above may be somewhat misleading. Throughout the code, the average function has one link to the specifications indicating what part of the specifications document is implemented by that function. One interesting difference that can be found if the files in the wavelet transformation module are examined, is that it has a lot more constants defined in the source code, all of which are linked to the location in the specifications document where they are defined.

However, the fact that the wavelet transformation module has so many links indicates that this is a type of project where HyperTrace could make code walkthroughs and debugging easier. Since all the constants and types defined in the specification are linked to the implementing code, they can easily be checked.

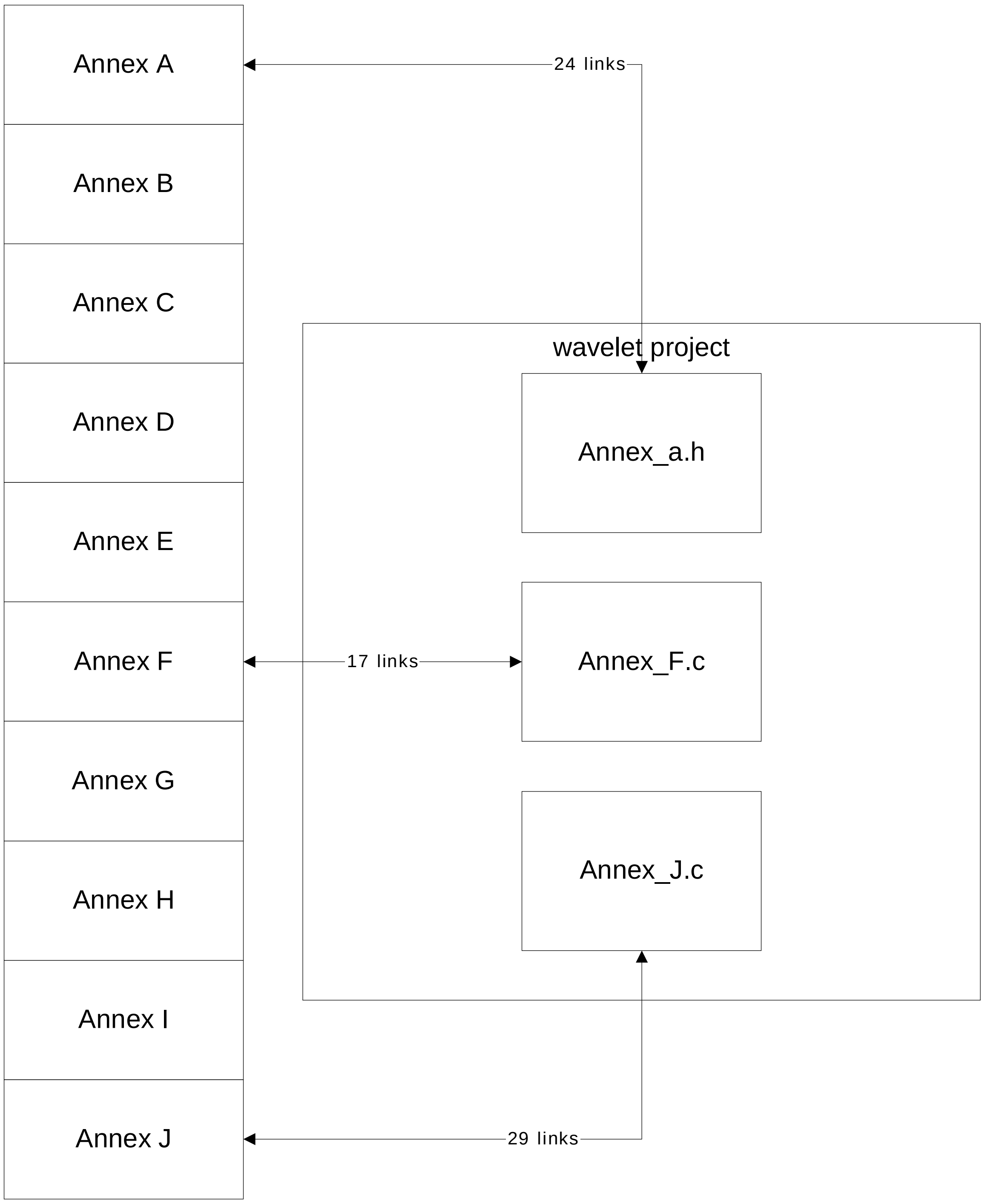


Figure 4.5: Links for the wavelet module

Figure 4.5 shows how the links for the wavelet module are distributed within the different source code files in the wavelet implementation module. As is evident in the figure, the module was written with the specifications clearly laying out how the implementation should work. This makes it easier to read the source code, since there is a clear connection between specifications and code.

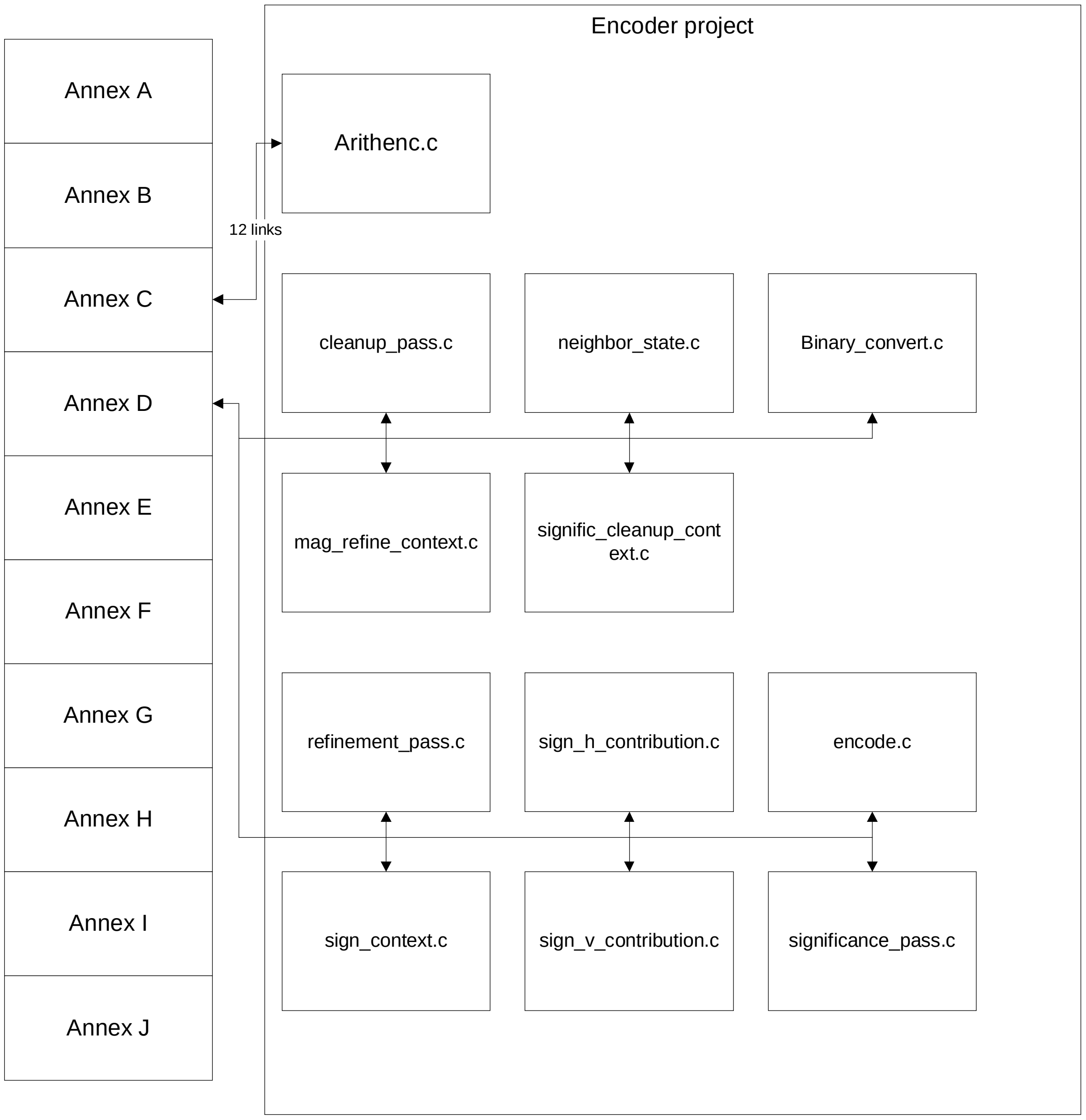


Figure 4.6: Links for the Encoder project

Figure 4.6 shows the distribution of links in the Encoder project. An interesting note on this is that for annex D there is one link for each source code file. This indicates that each file includes only one function (which is true and can be verified by looking at the source code files). This represents a different approach than that taken for the wavelet project, where each file contained the implementation of one annex in the specifications document. Annex C of the encoder project has all the functionality in one .c file, the same way used in the wavelet project.

This difference can be caused by relaxed coding standards, different authors and other factors in the project. The significance of the differing approaches can be minor, but if the team has coding guidelines set for the project, they should be followed. By using HyperTrace, this can be verified easily as an extra bonus.

Figure 4.7 shows the links and their distribution in the decoder project. As we can see, the decoder is similar to the encoder project in how it is split into one file for each function implemented (meeting one requirement from the specifications document).

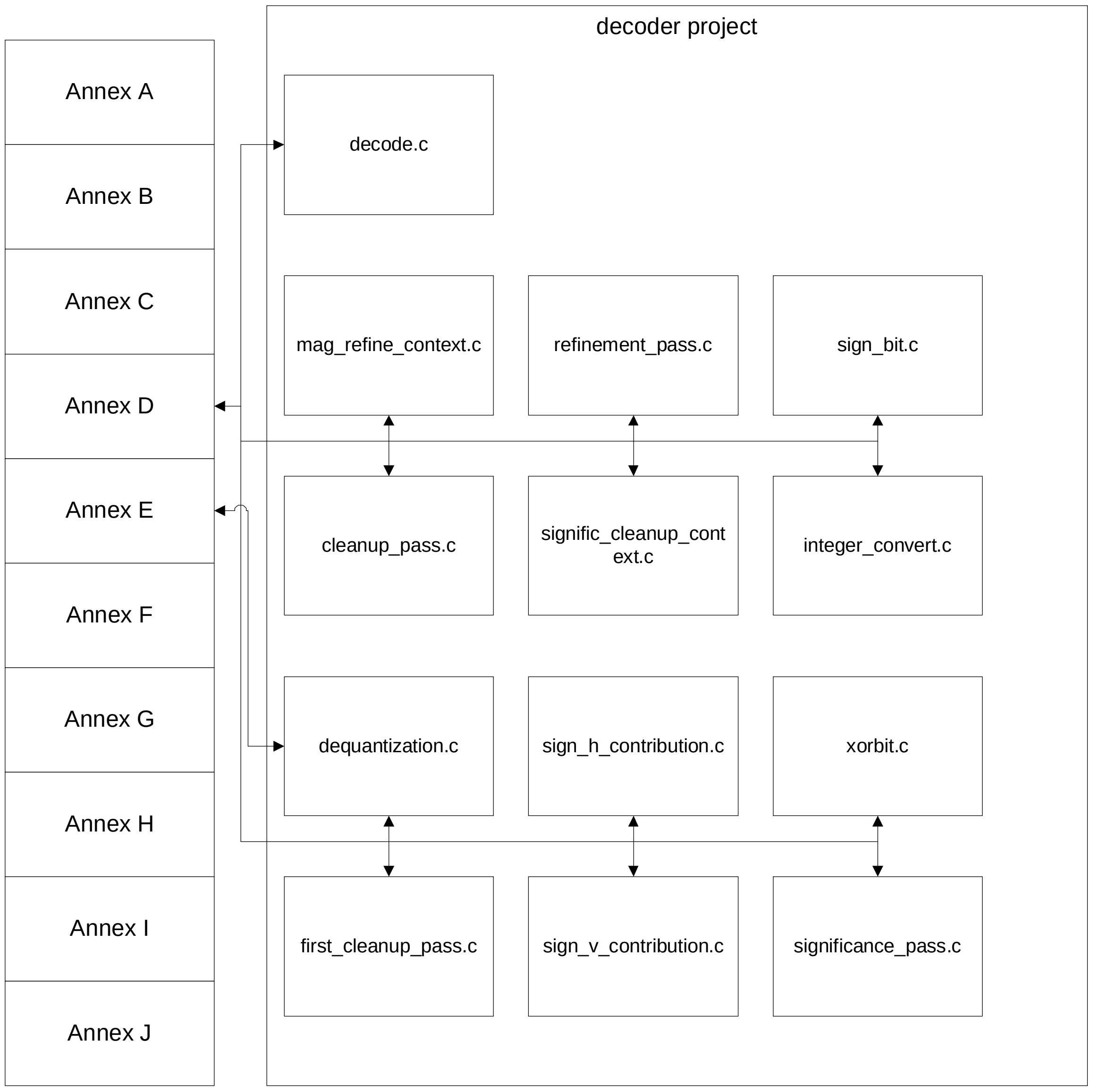


Figure 4.7: Links for the decoder project

One thing the statistics can provide is an indication of a set of parameters regarding the quality of the software. The different link/LOC numbers can indicate something about the software engineer who wrote the code, the quality and clarity of the specifications document and the complexity of the software built. Note that this is not a way to prove something about the software, but rather indicate where further examination might show things that otherwise would have remained undetected. As we can see in the statistics in figure 4.1, the links are fairly well contained within chapters. The wavelet module does not link to information relevant to the jpeg_encode module and vice versa. This can indicate that the layout and separation of modules in the specifications document is very good and the modules are not too interconnected. That the jpeg_encode and jpeg_decode modules share specifications in some parts is to be expected, since they are two sides of the same issue.

Typically, a module that has very few links can indicate that the code written for that module only loosely follows specifications, the work was sloppy (no links inserted) or the specifications are unclear or poorly written. However, it might also be simply be indicative of a very small and/or simple module, where many links are not required to create the understanding needed.

The JPEG 2000 project contains additional documents, like the SRS, but these mainly refer to the JPEG 2000 specification from the JPEG group and therefore do not introduce much new information.

4.8. Summary of examples

The JPEG 2000 example shows some of the strengths of HyperTrace. It confirmed that overhead for the user is minimal. There is not enough data to definitively determine whether the number of links is typical for a project of this type, but it can be inferred.

Work-division between functions in the code closely matches the division in the specifications.

4.9. Summary of chapter

Using hyperlinks for traceability leads to a solution that is user-friendly and easy to learn. The possibility of using the resulting links in calculating metrics that might be used to demonstrate completeness and complexity parameters of the project was also discussed. After studying a larger project with this tool, it became possible to reason about connections between parts of the specifications document and the code. Even if the metrics are not 100% reliable, it has at least been shown that the links can give a very good insight into which parts of the specifications are related to any module in the source code.

CHAPTER FIVE

5 CONCLUSION

5.1. Summary

HyperTrace enables traceability information to be gathered and used at a minimum cost to the user. The tool has been shown to be non-intrusive to the development environment, even to the point where the user can ignore it if it is not needed. While it does not supply the rigorous process supported by some of the other requirements engineering/life cycle tools on the market, HyperTrace provides clear benefits for people currently not using traceability tools at all. The type independence of hyperlinks is clearly available with the same semantics used for every type of document linked.

A look back at chapter three shows the following design goals:

- **Minimum administrative overhead:** Resulting implementation: The only overhead added by our tool is installation into Visual C++ and Word.

- **Minimum memory footprint:** Resulting implementation: Memory overhead is less than 6 MB total. When it comes to application storage overhead, all binary code for this project fits on a single floppy disk.
- **Ease of use:** Resulting implementation: Very easy to use.
- **Non-intrusive to developers:** Resulting implementation: Developers can ignore the tool if they don’t like it or don’t want to use it for a current project.

- **Ease of integration with new tools:** Resulting implementation: Using standard Hypertext concepts central to the World Wide Web makes integration easier by taking advantage of the Web's focus in most current applications.

Chapter 3 defined the conceptual architecture of HyperTrace and how it is implemented. In addition, it laid out the required operations needed by any application that is to be integrated with HyperTrace. Chapter 4 described the result of applying HyperTrace to two different examples, illustrating the simplicity of using the tool and the benefits it has.

The research for this thesis has dealt with the development of a hyperlinks based traceability tool and integration with commercial off-the-shelf components. Questions concerning whether or not and how this can be done have been partially answered and some foundations laid for integrating tools in the future.

5.2. Future work

5.2.1 Link Drawing/Visualization

To further the advantage of using the HyperTrace system, a Link Drawing/Visualization tool can be used. It would go through a set of documents, gather up all the links and create a map, showing what links to where in other documents. Having this type of tool can be beneficial, since it also allows quick traversal of the web created by the links.

The map that can be drawn by such a tool can be used to identify possible errors in the process that lead from the first document to the source code. One way this could be done is to check for missing links. For example, if a link can be followed from the initial requirements specifications to the design document but there is no link from that point to the source code, it is perhaps indicative that a link was not inserted when the source code was written or that the source code has not yet been written.

However, while this looks promising, emphasis must be placed on the fact that the results of such an examination of documents can only provide pointers. It does not in any way guarantee that the software has been written according to the requirements specification, but it can provide an insight into where things might be wrong.

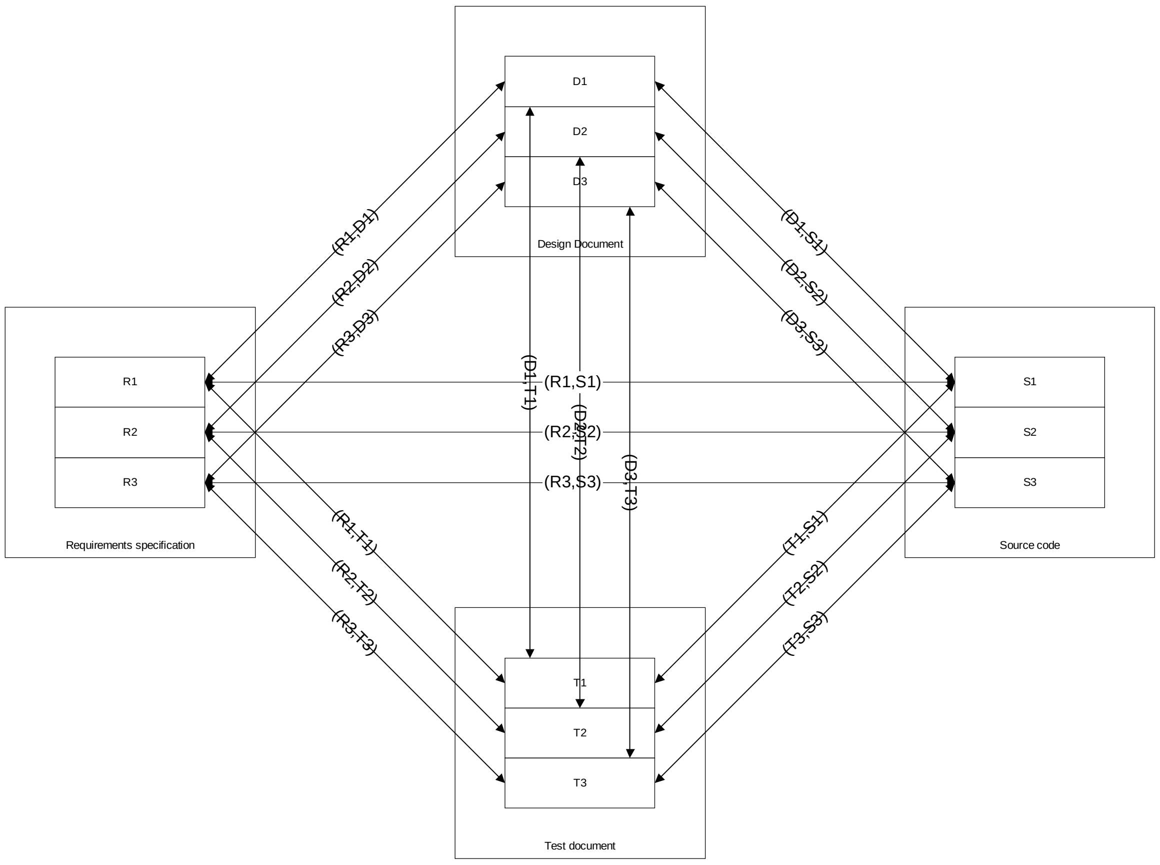


Figure 5.1: Link map

Figure 5.1 shows a simplified map of links, where each phase of the project (requirements, design, code and test) has three elements. For simplicity, the first element of each phase is linked to the first element of all the other phases. In reality, the map would be more unreadable than this, but it gives the general idea (a real map would not be so “complete” and simple).

5.2.2 Link Management

To further the use of hyperlinks, a link management tool can be used. Having a tool to manage links can be beneficial, in the following cases:

- Document locations change, links must be changed to keep up
- Source code locations change, links must be changed to keep up
- Help in detecting and inserting transitive links. Shortly described, there are link pairs (src, dest) (1,2), (2,3) indicating a possible missing link (1,3). This must be verified by a human, since there is no way of knowing for sure whether link (1,3) is missing or has been purposely left out for whatever reason. Figure 5.2 shows this in practice.

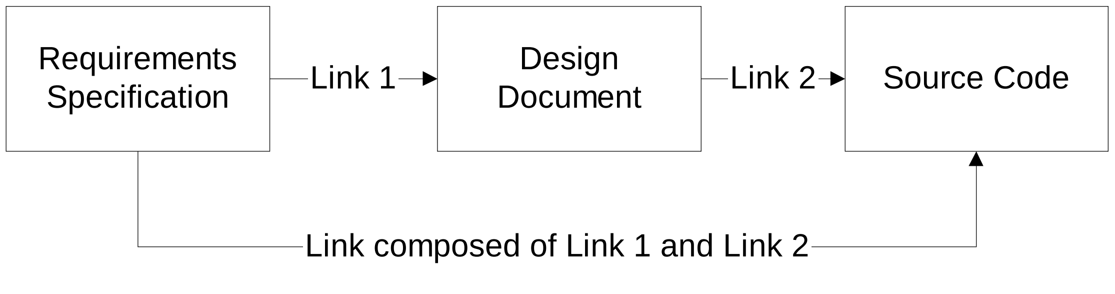

Requirements Specification
Link 1
Design Document
Link 2
Source Code
Link composed of Link 1 and Link 2

Figure 5.2: Composing links

- Help in detecting and inserting links that might be missing. Shortly described, the link pairs (src,dest) (1,3), (2,3) indicate that there could be a link (1,2) that is missing. This can only be verified through human intervention, but the tool could identify possible missing links by looking at the links and how they are related. Figure 5.3 shows an example of this.

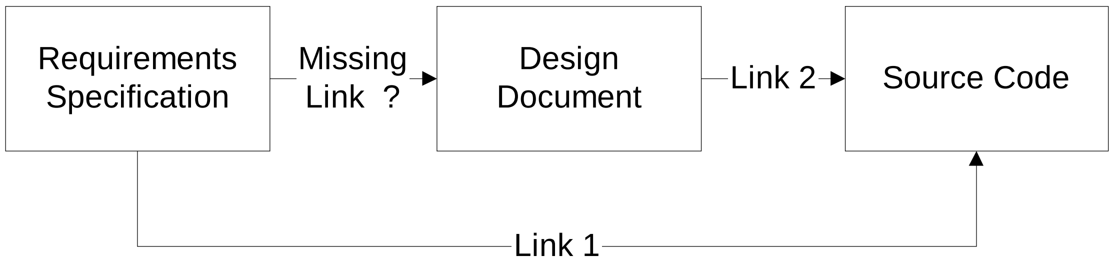


Figure 5.3: Inferring links from property of transitivity

- Check that links are valid. In some cases, parts of documents are deleted as part of the development process. However, there might still be links outside the document that point to information now deleted, and the anchor of the link will be missing. This can, by examining all the links for a project, be detected, but human intervention is required to correct the problem.

5.2.3 Traceability Matrix Generation

Another possible extension of the HyperTrace tool is a tool for automatic generation of traceability matrices. This would build on the tool already described for link visualization (mapping the entire project and the set of links). From this information, a table could be constructed, with a column for each document and the number of rows determined by the sets of links found.

Requirements.doc	Design.Doc	Test.Doc	Source code
R1	D1	T1	S1
R2	D2	T2	S2
R3	D2	T3	S2

Table 5.1: Traceability matrix

Table 5.1 shows a possible traceability matrix with links taken from Figure 6.1. As an added benefit, the links can be inserted to make the matrix a set of links organized in a table.

The rationale for building such a tool is that certain organizations need to follow stringent process descriptions and therefore need to generate a traceability matrix for each document they have written as part of a project.

5.2.4 Evaluating metrics

After inserting all the links into documentation, source code and other pieces of information related to a project, one could study the density of links and other related ratios. Further work needs to be done to establish which of these actually mean anything and if so, what they mean. Included here as example metrics are number of links / LOC, how well the links match modules (i.e. links from the same part in the specifications document should go to the same module in the implementation and so on). In chapter four, some ideas were proposed, but there was not enough data gathered for this research to verify those metrics. Studying many more projects would be required to establish correlations between the links inserted and the quality properties of the project.

6 BIBLIOGRAPHY

[Anderson 1994]: Kenneth M. Anderson, Richard N. Taylor and E. James Whitehead, "Chimera: Hypertext for Heterogeneous Software Development Environments", Proceedings of the 1994 ACM European conference on Hypermedia technology, 1994, Pages 94 – 107.

[Anderson 1999]: Kenneth M. Andersen, "Supporting industrial hyperwebs: lessons in scalability", Proceedings of the 1999 international conference on Software engineering, 1999, Pages 573 - 582

[Cavano 1987] Joseph P. Cavano and Frank S. LaMonica, "Quality Assurance in Future Development Environments", IEEE Software, September 1987, pp 26-34

[Chimera]: Ken Anderson, Jim Whitehead, Joe Feise, Yuzo Kanomata, Dick Taylor, "Chimera 2.0", http://www.ics.uci.edu/pub/chimera/index.html (August 1, 2000)

[Conklin 1988]: Jeff Conklin and Michael L. Begeman, "gIBIS: A Hypertext Tool for Exploratory Policy Discussion", ACM Trans. Office Information Systems, Oct. 1988, pp. 303-330

[Corriveau 1996]: J-P. Corriveau, "Traceability Process for Large OO Projects", Computer, 29(9), pp. 63-67, September 1996.

[Davis 1997]: Hugh Davis, Sigi Reich and Antoine Rizk, "OHP – Open Hypermedia Protocol, working draft 2.0 20th June 1997", http://www.ecs.soton.ac.uk/~hcd/ohp/ohp.htm, August 1, 2000.

[Deutsch 1998]: M. S. Deutsch and R. R. Willis, ”Software Quality Engineering: A Total Technical and Management Approach”, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[Dömges 1998]: Ralf Dömges and Klaus Pohl, “Adapting traceability environments to project-specific needs”, Communications of the ACM, Volume 41 , Issue 12 (1998), Pages 55-62.

[Dorfman 1984]: Merlin Dorfman and Richard F. Flynn, “Arts – An Automated Requirements Traceability System”, J. Systems and Software, 4(1), 1984, pp. 63-74

[Fickas 1995]: Stephen Fickas and Martin S. Feather, “Requirements Monitoring in Dynamic Environments”, Proceedings of the Second IEEE International Symposium on Requirements Engineering, IEEE Computer Society Press, York, England, March 1995

[Gillies 1992]: A. C. Gillies, “Software Quality - Theory and management”, International Thomson Computer Press, 1992.

[Gotel 1993]: Gotel, O.C.Z. & Finkelstein, A.C.W. (1993). “An Analysis of the Requirements Traceability Problem”, Technical Report TR-93-41, Department of Computing, Imperial College.

[Gotel 1995]: Orlenta Gotel and Anthony Finkelstein, “Contribution Structures”, 2nd IEEE International Symposium on Requirements Engineering, March 1995.

[Horowitz 1986]: Ellis Horowitz, Ronald Williamson: “SODOS: A Software Documentation Support Environment - Its Definition.”, IEEE Transactions on Software Engineering, Volume 12, Number 8, August 1986, pp 849-859

[IEEE12207]: The Institute of Electrical and Electronics Engineers, Inc. “Industry Implementation of International Standard ISO/IEC: ISO/IEC 12207 Standard for Information Technology Software Life Cycle processes”, IEEE/EIA Std 12207.0-1996

[IEEEGST]: The Institute of Electrical and Electronics Engineers, Inc. “Standard Glossary of Software Terminology”, IEEE Std 610.12-1990

[Jarke 1994]: M. Jarke, H.W. Nissen, K. Pohl, ”Tool Integration in Evolving Information Systems Environments”, 3rd GI Workshop Information Systems and Artificial Intelligence: Administration and Processing of Complex Structures, Hamburg, Germany, February 1994

[JPEG2000]: JPEG Group, “JPEG 2000 Image Coding System” , Final Committee Draft Version 1.0, http://www.jpeg.org

[Kaindl 1993]: Kaindl, H., “The Missing Link in Requirements Engineering”, ACM SIGSOFT Software Engineering Notes, Vol. 18, No. 2, pp. 30-39.

[Lindsay 98]: Peter Lindsay and Owen Traynor, “Supporting Fine-Grained Traceability in Software Developmen Environments”, Proc. 8th System Configuration Management Symposium, Springer Verlag LNCS 1439, 1998, 133-139.

[Liu 1989]: Lung-Chun Liu and Ellis Horowitz, “A Formal Model for Software Project Management”, IEEE Transactions on Software Engineering, Vol 15, No 10, October 1989.

[MILSTD498]: U.S Department of Defense, “MIL-STD-498”, Software Development and Documentation, Naval Publications and Forms Center, Philadelphia, Dec. 5, 1994

[Noll 1991]: John Noll and Walt Scacchi, “Integrating Diverse Information Repositories: A Distributed Hypertext Approach”, Revised version appears in IEEE COMPUTER, Vol.24 (12), December 1991, pp.38-45.

[Noll 1994]: John Noll and Walt Scacchi. “A Hypertext System for Integrating Heterogeneous, Autonomous Software Repositories”, Proc. 4th. Irvine Software Symposium, University of California, Irvine, CA, 49-60, April, 1994

[Oman 1990]: Oman, P. and C.R. Cook, "Design and Code Traceability Using PDL Metrics Tool", Journal of Systems and Software 12 (July 1990), pp. 189-198.

[Pinheiro 1996] : F.A.C Pinheiro and J. A. Goguen, “An Object-Oriented Tool for Tracing Requirements”, IEEE Software, March 1996, pp. 52-64.

[Platt 1999]: David S. Platt, “Understanding COM+”, Microsoft Press 1999.

[Rama 1990] : C. V. Ramamoorthy, Yutaka Usuda, Atul Prakash, W. T. Tsai: “The Evolution Support Environment System”, IEEE Transactions on Software Engineering, November 1990: 1225-1234 (1990)

[Ramesh 1995]: Bala Ramesh, Lt. Curtis Stubbs, Lt. Cmdr. Timothy Powers, and Michael Edwards, “Lessons Learned from Implementing Requirements Traceability”, STSC Crosstalk, April 1995

[Ramesh 1998]: Balasubramaniam Ramesh, “Factors influencing requirements traceability practice”, Commun. ACM 41, 12 (Dec. 1998), Pages 37 – 44

[Rogerson 1997]: Dale Rogerson, “Inside COM”, Microsoft Press, 1997.

[Scach 1996]: S. R. Schach, “Classical and Object-Oriented Software Engineering”, Irwin, 1996.

[Shepherd 1997]: George Shepherd and Scot Wingo, “The Visual Programmer”, Microsoft Journal, October 1997.

[Shepherd 1998]: George Shepherd and Scot Wingo, “The Visual Programmer”, Microsoft Journal, October 1998.

[Sommerville 1995]: I. Sommerville, “Software engineering”, Addison-Wesley Publishing Company, 5th edition, 1995.

[Whitehead 1997]: E. James Whitehead, Jr., “An Architectural Model for Application Integration in Open Hypermedia Environments”, Proceedings of Hypertext'97, the Eighth ACM Conference on Hypermedia Technology, pages 1-12.

APPENDIX A : INSTALLATION INSTRUCTIONS

Installation

The installation of HyperTrace is done in three stages, installing into Visual C++, installing the link exchanger and installing the word macros.

Visual C++

User who wants to use the addins must be member of administrators group (local administrator) to be able to load the .dlls into the environment.

1. Copy the .dll files (hotlinks.dll and insertinwardslink.dll) to some directory (location not important)
2. In Visual C++, open the customize dialog (under tools menu)
3. Under add-ins and macro files click on browse
4. For each of the two dlls, add them to the list, making sure that the following are installed (and checked in the list)
    a. Hotlinks Code Linker
    b. InsertInwardsLink.DSAddIn.1
5. A toolbar will appear (for the link insertion module)

Link Exchanger / Link traversal

For the link traversal (traversing links into Visual C++ code), the VstudioLinker.exe program must be running. Therefore, copy this file onto the local machine and create a shortcut to it in the startup folder for the users needing this program.

For the link exchanger, copy the BSTRexchange.exe program onto the local machine. Then execute the command “BSTRexchange.exe /RegServer”, which will register the ATL COM object with the environment.

Word Macros

Copy the macros from the file “macros.doc” into the macro editor and store them in the template used for writing documents. Make the InsertLink and InsertBiLink macros available from the toolbar (toolbar customization in Word) and you can also assign shortcuts to them. Enter the macro editor, and when looking at either of them, click tools, references and add BSTRexchange library to the list of references.

Removal of HyperTrace

The removal of HyperTrace is done in three stages, as for the installation.

Visual C++

1. Go to the tools menu, choose customization
2. From the addins/macros page, deselect Hotlinks Code Linker and InsertInwardsLink.AddIn.1
3. Delete the DLLs from the machine.

Link Exchanger / Link traversal

Delete VstudioLinker.exe from the location it was copied to and from the startup menu.

Execute the command “BSTRexchange.exe /UnRegServer”, then delete BSTRexchange.exe

Word Macros

Remove the macros from the environment, with it any shortcuts or toolbar items associated with the macros.

APPENDIX B: HYPERTRACE SPECIFICATIONS DOCUMENT

INTRODUCTION

Purpose of this document

This document serves as specification and as an example of linking the within the project artifacts. It is part of a research project into traceability in software engineering and will lay out the details for a tool that will enable users to hyperlink their source code, and other artifacts in ways not yet in widespread use. This is much like the way the world wide web works, so users will se a familiar way of browsing the information.

Scope of product

The purpose of this specification is to specify the details to be implemented for the traceability tool, what the major components will be and how they work together to do two-way linking.

For further information about the project, see the reference document.

For a general idea, we imagine traceability links possibly between every single artifact of a project.

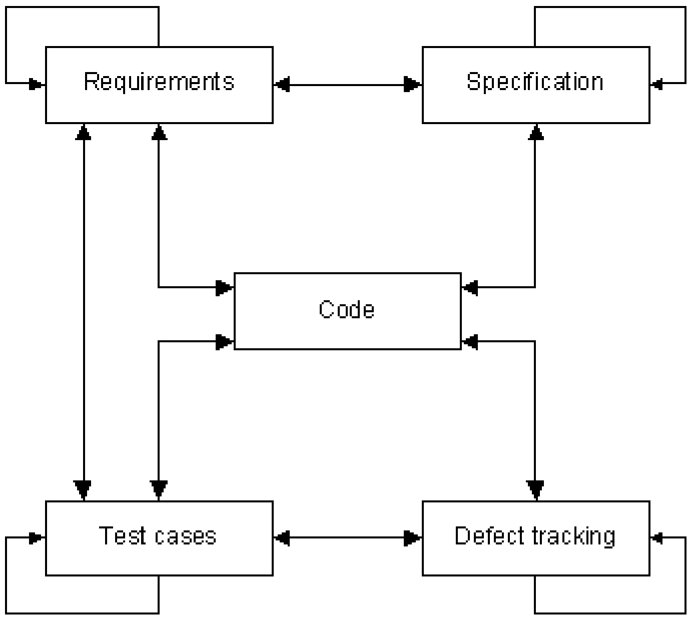


Figure 4: Traceability link illustration

DEFINITIONS, ACRONYMS, ABBREVIATIONS

- COM: Component Object Model
- ATL: Active Template Libraries
- OLE: Object Linking and Embedding
- Visual C++: Microsoft ® Visual C++® 6.0
- WIN32 API: Microsoft ® Windows 32 bit API
- http: hypertext transfer protocol
- Automation: programmatically making an application perform some preprogrammed operation on behalf of the user
- .cpp: C++ source code file

- .h: C/C++ header file
- .dsw/.dsp: Visual C++ project management file.

REFERENCES

1. Geir A. Bjune and Jack R. Hagemeister, "Definitions of traceability", January 2000.
2. Rfc1945, "Hypertext Transfer Protocol -- HTTP/1.0", May 1996
3. Q247035 - HOWTO: Automate Visual C++, Microsoft Product Support Services

OVERVIEW OF DOCUMENT

The document is organized according to standards set in Cpt S 580, section 1. Section 2 is a general description of the product; section 3 is a specification of the different elements of the system and how they all interact.

GENERAL DESCRIPTION

Product perspective

The specification specifies a system to enable traceability links to and from documents written in Microsoft® Office, Visual C++® and other tools capable of doing hyperlinks. The purpose of the software is to enable users to more easily and quickly locate documentation and code related to some artifact that they have access to on their computer.

Static traceability matrices don’t offer much in ways of allowing the user to easily locate a piece of code related to a requirement, test case or bug database entry. By using hyperlinks between the different information elements, the user can simply click on a link to go to the related information.

Architecture concerns

Even though the idea of the system can be extended to just about any development system in use today, we have chosen to study this using Microsoft tools for several reasons:

- Configurability through plugins
- COM/OLE interfaces enabling remote control and automation of applications
- They are in widespread use, making it easier to show how the tool works and uses technology that users are familiar with. See reference 1 for a discussion about this.

PRODUCT FUNCTIONS

The different components together tie together different applications. For linking within the Office environment the user can rely on the built in technology to support this. This means that there will be a way for the user to click on a link in for example Word and jump to a related piece of source code in Visual C++.

General constraints

The software product is more a technology demonstrator than a stable production tool, so some shortcuts may be taken in the implementation if it still delivers the same basic functionality.

SPECIFIC REQUIREMENTS

Link to Visual C++

Hyperlinking to the source code

Link to this location: C:\demo\doc\ToolSpecification.doc#l949920380

Outbound link: Link to Code – Parsing the URL

Outbound link: Link to Code – Invoking DevStudio

Since Visual C++ by itself has no mechanism for supporting hyperlinks, this will have to be done using an external program or an add-in. Whether or not it is a plugin is an implementation issue, the only effect it will have is whether or not Visual C++ will have to be running at the time of traversing a hyperlink.

Input: The hyperlink will be specified as follows:

Project=<absolute path to C++ .dsw or .dsp file>enabling Visual C++ to open the project.

Filename=<Relative reference to the file (.cpp/.h etc) enabling opening the right document

Target=<number> where number is the identifier of the link inserted into Visual Studio.

Source code contents: To find the link, the project must exist; the filename must exist within that project and contain the following text:

```
// link=number
```

For example:

// link=9502 in file commands.cpp in project c:\testproj.dsw will be specified as

http://localhost:90/project=c:\\testproj.dsw&filename=commands.cpp&target=9502

The software component that takes these requests will have to listen to a TCP port (e.g. 90) and parse the request according to http specifications. The component then will use COM to access the Visual C++ object model and perform the required actions, opening the project, file and then finding the location being linked to.

Inserting hyperlinks in the source code

Link to this location: C:\demo\doc\ToolSpecification.doc#l949930672

Outbound link: Link to Code – Insert a linkable point in the code

For simplicity of use, a component to insert the "// link=number", and also what the corresponding hyperlink will be, into the Visual C++ source code. The identifier used is the time, given in number of seconds since January 1st, 1970 (standard time_t format). The fundamental assumption here is that no two links will be added to the same file with the same identifier. This code needs to be in an add-in so that the user can invoke it using some standard Windows-approach, like a toolbar button or a menu item.

The result is:

```
// link=954564856
// http://localhost:90/cgi?project=F:\\geir\\code\\InsertInwardsLink.dsp&filename=InsertInwardsLink.cpp&target=954564856
```

If the user has selected some part of the active document before doing this, the inserted result will look like this:

```
// link=954564888
// http://localhost:90/cgi?project=F:\\geir\\code\\InsertInwardsLink.dsp&filename=InsertInwardsLink.cpp&target=954564888

// ***** Begin Code for link: *****

#ifdef _DEBUG

void GetLastErrorDescription(CComBSTR& bstr)
{
	CComPtr<IErrorInfo> pErrorInfo;
	if (GetErrorInfo(0, &pErrorInfo) == S_OK)
		pErrorInfo->GetDescription(&bstr);
}

#endif //_DEBUG



// ***** End Code for link: *****
```

This will enable the user to specify exactly what a link points to (semantics of a link location). The second line of the inserted text is the hyperlink, which can then be copied into other programs and used as a hyperlink to the line above.

Linking from Visual C++

Link to this location: C:\demo\doc\ToolSpecification.doc#l949934807

Outbound link: Link to Code – code for actiovation of all types of links

The other half of the linking is the linking out from Visual C++. Since the source code has to be semantically intact for the compiler to be able to use the code, the links have to be embedded in comments. By using a well-known keyword in the comment, an add-in to Visual C++ can take the link and open it.

That way, the line will look like this:

// $<item_type>: link target

The link target is defined as follows:

filename#bookmark

e.g.

c:\\project\\documents\\specification_mainsystem.doc#introduction

The use of filename#bookmark is a common element to both Microsoft® Office and also used on the World Wide Web, so links to HTML documents on remote servers look similarly.

In the future, XML might be considered to replace the identifiers in this section, if feasible. Opening the link itself will be done using shell-execution of the document file being opened, so that the users settings w.r.t applications to use will be respected. There are three subcases of this:

Linking to design/specification documents

Link to this location: C:\demo\doc\ToolSpecification.doc#l949931005

Outbound link: Link to Code

Documents that specify some functionality present in the source code will use a $spec identifier, so that tools to parse this can be developed at some later point.

Example:

```
// $spec: c:\\project\\documents\specification_subsystem_1.doc#130
```

This will fetch the document name from the module keeping track of where the link was copied from (in Word) and paste this into it. It will also insert a linkable line in this file so that there can be a bidirectional link.

Linking to test cases/defect tracking documents

Link to this location: C:\demo\doc\ToolSpecification.doc#l949934600

Outbound link: Link to Code - Parsing of $test: <link>

Documents that specify some test cases or defects found in the source code will use a $test identifier, so that tools to parse this can be developed at some later point.

Example:

```
// $test: c:\\project\\documents\specification_subsystem_1.doc#130
```

This will fetch the document name from the module keeping track of where the link was copied from (in Word) and paste this into it. It will also insert a linkable line in this file so that there can be a bidirectional link.

Linking to general documents

Link to this location: C:\demo\doc\ToolSpecification.doc#l949934891

Outbound link: Link to Code – Parsing of $doc: <link>

Documents without a clear meaning (for extensibility) are labeled $doc, since that just identifies some information element.

Example:

```
// $doc: c:\\project\\documents\specification_subsystem_1.doc#130
```

This will fetch the document name from the module keeping track of where the link was copied from (in Word) and paste this into it. It will also insert a linkable line in this file so that there can be a bidirectional link.

Automatically inserting bidirectional links

Link to this location: C:\demo\doc\ToolSpecification.doc#l949934959

Outbound link: **Link to Code** - Inserting Link into VC++ code and sending link to StringExchange object

Link to this location: C:\demo\doc\ToolSpecification.doc#l949935048

Outbound link: **Link to Code** – Link to StringExchange object source code (the entire object)

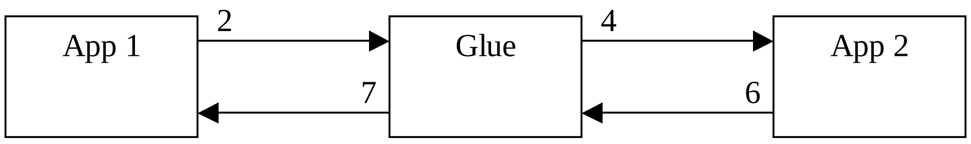
Purpose

To make the process of inserting bidirectional links easier for the user, there will be functionality that will let the user insert a from-link in one document and a to-link in another document and the reverse link will also be added, regardless what types of documents we are talking about (Word, source code, spreadsheet, webpage)

Functionality

```
1: app1 inserts its internal link (link1) (what app2 will link to) in the
     document (link-destination)
2: app1 writes this string (link1) to some shared location (named pipe or
     using sockets)
```

```
3: app1 keeps retrying to read link2 until success

4: app2 reads the link (link1) that app1 wrote to shared location and inserts

       a link (link1) in its own document pointing to app1's document

5: app2 inserts a link (link2) to location within its document

6: app2 writes that link (link2) to the shared location that app1 is waiting

       for it to arrive on

7: app1 reads link (link2) from shared location and inserts it into document

8: app1 and app2 keeps running like nothing happened
```


App 1
2
Glue
7
4
App 2
6