Proceedings of the 9th Working Conference on Reverse Engineering (WCRE 2002), IEEE Computer Society Press, 29 Oct – 1 Nov, Richmond, Virginia, 2002.

Generation of Architectural Documentation using XML

Claudio Riva and Yaojin Yang
Nokia Research Center
Software Architecture Group
P.O. Box 407, FIN-00045 NOKIA GROUP
Tel.: +358 50 483 7403, Fax.: +358 7180 36308
{claudio.riva | yang.yaojin}@nokia.com

Abstract

Documentation generation is the process of creating the system documentation at different levels of abstraction from the source code for a legacy system. The main goal is to help the stakeholders to understand the system with representations at the appropriate abstraction level. We follow an architecture reconstruction method to create an architectural model of the system and we use a documentation generation technique to re-document its architecture. In this article, we present our XML based approach for generating the architectural documentation of a software system.

Keywords: Documentation generation, XML, reverse engineering, XSLT

1 INTRODUCTION

The software documentation is an essential part of a software system. Information that describes the subject software system in any kind of format and at any level of abstraction can be considered documentation. Documentation helps the stakeholders to understand and to maintain the system. The source code is the most detailed and reliable documentation, but usually it is not feasible and necessary to comprehend all the details in it. If we can create an abstract view of the source code, then we have the capability of zooming on specific details starting from the top level [19].

Documentation generation is one of the key activities of system reverse engineering that is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction [3]. A. van Deursen et al. indicate that a flexible documentation should adhere to four criteria:

- Documentation should be available on different levels of abstraction.
- Documentation users must be able to move smoothly from one level of abstraction to another, without loosing their position in the documentation.
- The different levels of abstraction must be meaningful for the intended documentation users.
- The documentation needs to be consistent with the source code during the system lifecycle.

In our previous works [16][2] [15], we have proposed an architecture reconstruction method that allows us to re-construct the architecture of a software system (with static and dynamic views) from the implementation. The output of the reconstruction is a high level model that describes different architectural aspects of the system (such as the component view, the task view and the feature view). We use the output of the reconstruction to re-document the architecture of the system.

In this article, we describe our approach for the generation of the architectural documents. The process consists of four phases: *concept definition* (to identify the architecturally relevant concepts that have to be presented in the documentation and their mappings to the implementation), *documentation extraction* (to extract the elements of the documentation by a lexical and syntactical analysis of the source code), *documentation abstraction* (to classify and organise the documentation in a hierarchical format and derive more abstracted views) and *documentation presentation* (to present the documents in a human readable format). The main trigger of our work is to enable the software architects to re-document the architectures of large software systems, as presented in our previous work [15].

In Section 3, we give an overview of the architecture reconstruction method. In Section 4, we describe our requirements for the documentation of a software architecture. In Section 5, we introduce key technologies that we use in our approach. In Section 6 and 7, we respectively describe our approach and the supporting environment. In Section 8, we demonstrate the approach with a simple example. In Section 9, we make some observations on the approach and in Section 10 we draw the final conclusions.

2 RELATED WORK

In our previous work, we have related our work with other research in the field of architecture reconstruction [15] and dynamic analysis [2]. In this section, we report the major shortcomings that we have detected concerning the task of architecture re-documentation.

Kazman et al. propose an iterative reconstruction process based on the Dali workbench [8][7]. Dali allows the user to create a source code model in a SQL database. The user can then base the abstraction process (mainly a grouping activity) on a set of queries executed in the database. The reconstructed architecture is visualised using hierarchical graphs in Rigi.

Finnigan et al. [4] propose the Software Bookshelf that is a collection of tools for generating software architectures from program sources and presenting them in a Java-based web user interface. The goal is to keep the architectural documentation up to date. The tool has been used to extract the software architecture of Linux operating system [1]. One key feature is the web interface that allows the architects to publish the architectural diagrams on the intranet. The major difference between PBS and our approach is that (1) we generate UML diagrams that typically used by the system designers and (2) our environment is language and domain independent.

A. van Deursen et al. [19] propose a framework to generate documentation from source code level. It allows the user to create documentation at different abstraction levels. This framework includes three phases of source code analysis, extracting documentation and presenting documentation. Source code analysis provides several means for indicating constructs of interest in the source code. Extracting documentation is the phase to extract documentation concepts according to different abstraction levels. Presenting documentation is to build the interface between user and the documentation. Our approach refers to this framework, and further emphasises documentation abstraction by importing an architecture reconstruction environment to achieve architectural documentation generation. At the same time, our approach indicates an explicit way of how to integrate a structural data environment into documentation generation by using GXL technique.

Nentwitch et al. [12] have proposed a tool for publishing UML documents on the web. The tool does not address the specific needs of re-documentation but is based on open standards like XML, XMI, UML and SVG for visualising UML diagrams in a web browser.

Gail et al. [11] propose a reconstruction technique based on the reflexion models. The user starts with a structural high-level view model that is iteratively refined to rapidly gain knowledge about the source code. The technique is based on the definition of a set of mappings between the source code and the high-level concepts. Our architecture reconstruction method generalises this idea enabling the user to define any kind of mappings or transformation of the source code model.

3 ARCHITECTURE RECONSTRUCTION

The description of the software architecture should communicate the essential decisions that have been taken during the design of the software system [13]. The decisions are about the *concepts* of the system (the way we think of a system, its architectural style), the *architecturally significant requirements* (the major concerns that have to be addressed by a proper software architecture), the *structure* (the components and their relationships at the right level of abstraction) and the *texture* (design decisions at the implementation level that are architecturally relevant, such as design patterns and policies). Multiple views (such as the “4+1 model” proposed by Kruchten [9] and the architectural views proposed by Hofemeister [6]) are a practical way to effectively communicate the different aspects (and concerns) of the software architecture. In both the methods, static and dynamic aspects of the architecture are addressed.

Architecture reconstruction (or reverse architecting) concerns with the task of recovering the past design decisions that have been taken during the development of a system and to present them through multiple architectural views. It is a reverse engineering activity that has to infer the architectural rationale from the available artefacts created by the original developers (such as source code, design diagrams, user guides). The natural evolution of a software system also introduces new aspects that a reconstruction process can unveil.

The output of the reconstruction is a set of architectural views focused on particular concerns. In our experience, we have found useful to deliver the following architectural views:

- Conceptual view: describing the key architectural concepts that build the system. The instances of these concepts are presented in the other views.

- Component view: describing the major components, their interfaces and their logical relationships.
- Development view: describing the organisation of the source code files and their relationships (for example, include dependencies).
- Task view: describing the task allocation of the architectural entities and showing the inter task communications.
- Feature view: describing the run-time implementation of the features at a high level of abstraction.

The views are based on static aspects (captured without running the system) and dynamic aspects (concerning with the run-time behaviour). They are both necessary for the architectural description and they have to be adequately reverse engineering from the implementation.

We stress the point that only the correct choice of the architectural concepts to examine can deliver a meaningful high-level model to the architects. The architectural concepts are first class entities in the reconstruction process from the very early stages.

In our approach [16][2], we follow a four-step iterative process: *(1) definition of architectural concepts, (2) extraction, (3) abstraction and (4) presentation*. The goal of the first phase is to recover and clarify the architecturally significant concepts that build the system (building blocks and communication infrastructure). These concepts represent the way developers think of a system and they should become the terminology of the reconstruction process (for examples, applications, servers and software busses for a distributed system, while processes, queues, shared memories might be used for an operating system). The second phase builds a model of the system by extracting the relevant information from the implementation. The entities of the model are instances of the concepts identified in the previous phase. A correct choice of the concepts will ensure that the model is filled with entities at the right level of abstraction. For the static information, we rely on the source code and the system documentation. For the dynamic data, we instrument the system and trace its execution by simulation. In the third phase, we enrich the model with domain specific knowledge that is necessary to create a high level view of the architecture. This is a reasoning activity where we need to infer new logical and more abstracted relationships. In the last phase, we present the architectural views with effective visualisations that allow us to communicate the architectural information to the architects: hierarchical oriented graphs, web documents, UML, logical diagrams, message sequence charts and collaboration diagrams.

4 ARCHITECTURAL DOCUMENTATION

The description of the software architecture can be accomplished only by using a clear notation, which is semantically well defined and easy to understand by all stakeholders. The currently available architectural description languages (ADLs) have not spread in industry mainly because they are not generic enough, are not standardized and are poorly supported by tools. Several ADLs support descriptions for components, connectors, configurations, and/or other aspects of software architecture [10]. These ADLs have demonstrated various research ideas and they have limited scope. Most of the ADLs can only be used to describe one particular architectural view and have to be augmented with other modelling mechanisms. The big gap from the design and implementation languages used in software development together with the lack of good supporting tools have limited the use of ADLs in industry.

UML as a general-purpose design notation is an alternative to the current ADLs. UML is a standard now and most of our designers can understand it and use it. UML descriptions of software architecture not only provide a standard definition of the system structure and system terminology, but also facilitate consistent and broader understanding of the architecture and enable more extensive tool support for architecture design.

UML is a standard, but its current semantics fails to meet the criteria stated above: it is weak at describing interfaces, the abstractions it provides are not univocal and it provides little support for modelling architecturally significant information. UML is also not suitable for modelling reverse engineered architectural models: the notation does not allow to model all needed implementation concepts, and does not offer other support for typification than extension. UML as such is not an ideal language for describing the basic building blocks of software architecture (components, connectors and architectural configurations).

In [17] we have highlighted that architecture modelling demands a more precise notation, with improved support for interface modelling, a limited number of possible relationships and more rigorous semantics. For example, we have identified the following UML elements necessary for modelling the component view: *package, component, interface, generalisation, dependency and generalization*.

This situation poses several challenges when we need to present the reconstructed architecture to the developers. We do believe that the graph-based representations well suit the needs of the architecture reconstruction process but those representations are not very effective for the task of re-documentating the architecture. We have identified a set of key requirements that have to be satisfied:

- the format has to be to be simple to use and generally understood by developers. Graph based representations (e.g. the Rigi graphs) are not so intuitive for non-experts. The UML standard is the best candidate because it is universally used and understood, although its semantics is not precise.

- the internal Web is widely used for distributing documentation in the organisation. This suggests that the software architecture diagrams can be published on the web so that developers can search and browse documents with the web browser (similarly to the approach of the Personal Bookshelf [4])

- software architecture concerns also with the organisation and the classification of the system artefacts. Therefore, we need an elegant interface for showing the different categories (e.g. subsystems) and views (architectural views).

- the re-documentation process has to be language and domain independent similarly to our architecture reconstruction method.

5 XML, GXL AND XSLT

XML (Extensible Mark-up Language) [21] is a mark-up language for documenting structured information. An XML document, which is a plain text file, contains tags, values, and the relationships between those tags. XML is an ideal storage format for software documentation. This language has native support for structured information, which exactly software documentation is. Therefore, in an XML document, XML tags can be used to identify the software concepts, XML values can be used to record the facts of each concept, and relationships between those tags can be used to present the links between different concepts. XML also provides an independent platform for data exchange, because it does not adhere to any specific data format. The same XML file can be accepted by any XML supported application and can also be converted to different data formats using standard parsers or a transform language. Documentation generation can benefit from this feature to easily accomplish that the same content but has different presentations, so the efficiency of documentation generation is really improved. Moreover, XML is a plain text format, which is very easy to create.

GXL (Graph eXchange Language) [5] is a standard exchange format for graphs. It has been designed to support the interoperability between graph-based tools and in particular reverse engineering tools. GXL allows us to store hierarchical attributed typed graphs where nodes can represent concepts of documentation and edges can represent the relationships between these concepts.

XSLT (Extensible Stylesheet Language Transformation) [23] is a language for transforming XML documents into other XML documents or other formats. The transformation is achieved by defining a set of templates that are matched in the source XML tree and instantiated in the result tree. XSLT provides us with a simple and efficient way for processing XML documents.

6 OVERVIEW OF THE APPROACH

Our approach for documentation generation consists of two phases. In the first phase (consisting of *concept definition, documentation extraction* and *documentation abstraction*), we extract an architectural model of the system that we want to re-document. In the second phase (*presentation)*, we generate its documentation in different formats.

Concept definition. The first step concerns with defining the architectural concepts that have to be presented in the documentation. This requires understanding the stakeholders' interests in the architectural documentation. Discussions with the experts (architects, designers, programmers and testers) and the existing documents about the system are the main sources of information. The output of this phase is the definition of the concepts that we be described in the documentation and the mappings of these concepts to the implementation constructs.

Documentation extraction. We identify the implementation constructs for each concept that we have previously defined. We conduct (1) a lexical analysis for the concepts that are implemented semantically using existing tools (e.g. SourceNavigator [14]) and (2) a syntactical analysis to recover the concepts that have a standard syntax in implementation using regular expressions (e.g. Perl scripts). The extracted information is refined in a set of relational facts that are stored in a GXL file.

Documentation abstraction. The abstraction phase aims at creating a high level, architecturally significant documentation of the system architecture. This is mainly achieved by inferring new information, classifying the elements of the documentation in high-level categories and creating the different architectural views. The concepts that do not have a direct mapping to the implementation have to be identified during this phase. The resulting abstraction rules are mainly derived by the existing design documents and by interviews with the stakeholders. We represent the relational data about the architecture and the abstraction rules in terms of Prolog facts as we have presented in [15]. The gruoping rules are kept in separate files and are manually maintained by the architects. This procedure relies mainly on human experience and manual reasoning with the aid of

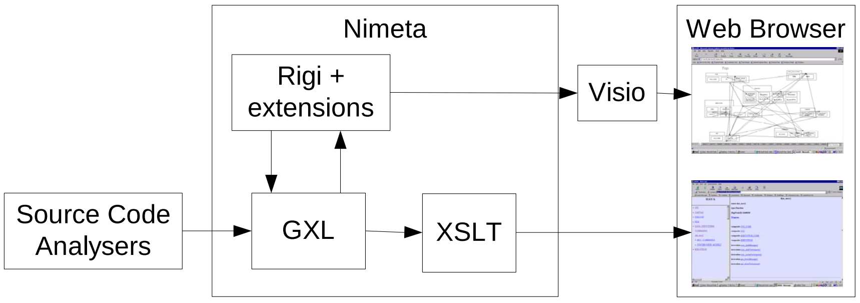

Figure 1. The architecture of the environment.

particular tools. Documentation abstraction is iterated until an appropriate level of abstraction has been reached.

Documentation presentation. The goal of this phase is to present the documentation in a human-readable format containing text and graphics. The main goal is to provide a user-friendly interface to the documentation that allows the users to navigate among the different levels of abstraction and the different parts of the documentation. The documentation is presented according to the different architectural views. To support the navigation of the documentation we use hyperlinks.

7 THE ENVIRONMENT

To support the architecture reconstruction and re-documentation process, we have integrated a set of tools in a reverse engineering environment called Nimeta [15] [2]. The architectural information is stored in a GXL repository that acts as a central storage point. The user interface is based on the Rigi environment [18] and we programmed in Tcl/Tk our own extensions. We have also integrated a Prolog engine to support the abstraction process [15]. XSLT [23] scripts are used for extracting the data from the GXL repository and converting to different formats. The generation of the architectural documentation is carried out by a set of XSLT scripts that convert the GXL data to the Visio [20] and HTML formats. The Figure 1 depicts the architecture of the environment.

The reconstruction environment supports two different formats of visualisation: Visio diagrams and HTML pages.

Visio [20] is a drawing tool that is widely used for creating UML diagrams. It supports XML and can also generate an HTML version of the diagrams. We use Visio to create UML-like diagrams, to calculate the layout and to generate the HTML version of the diagram. The Visio XML file is generated by a Tcl script that traverses the Rigi graph and calculates the shape, position and size of the UML elements. The script also organises the Visio

```
<xsl:template match="gxl/graph/node">
            <xsl:text>level Root </xsl:text>
            <xsl:value-of select="attr[@name=&quot;name&quot;]/string"/>
            <xsl:text>&#10;</xsl:text>
            <xsl:apply-templates/>
</xsl:template>

<xsl:template match="node/graph/node">
            <xsl:text>level </xsl:text>
            <xsl:value-of select="../../attr[@name=&quot;name&quot;]/string"/>
            <xsl:text> </xsl:text>
            <xsl:value-of select="attr[@name=&quot;name&quot;]/string"/>
            <xsl:text>&#10;</xsl:text>
            <xsl:apply-templates/>
</xsl:template>

<xsl:template match="edge">
<xsl:value-of select=" attr[@name=&quot;type&quot;]/string"/>
<xsl:text> </xsl:text>
<xsl:value-of select="//node[@id=@from]/attr[@name=&quot;name&quot;]/string"/>
<xsl:text> </xsl:text>
<xsl:value-of select="//node[@id=@to]/attr[@name=&quot;name&quot;]/string"/>
<xsl:text>&#10;</xsl:text>
</xsl:template>
```

Figure 2. The XSLT script to convert from GXL to RSF.

```
<xsl:template match="node" mode="detail">
    <xsl:if test="attr[@name='type']/string='Function'">
        <a name="{@id}"></a>
        <a name="{attr[@name='name']/string}"></a>

        <!--print general information-->
        <center><h3><xsl:value-of select="attr[@name='name']/string"/></h3></center>
        <xsl:for-each select="attr"> <p><b><xsl:value-of select="@name"/>:<xsl:value-of select="string"/></b></p>
        </xsl:for-each>
        <p><b><a href="pdf_hava/{@id}.pdf" target="diagram">Diagram</a></b></p>

        <!--the end of printing general information-->
        <xsl:variable name="currentNode_id"><xsl:value-of select="@id"/></xsl:variable>
        <xsl:call-template name="print_edge_from"> <xsl:with-param name="node_id" select="$currentNode_id"/>
        </xsl:call-template>
        <xsl:call-template name="print_edge_to"> <xsl:with-param name="node_id" select="$currentNode_id"/>
        </xsl:call-template>
    </xsl:if>
  <xsl:apply-templates select="graph/node" mode="detail"/>
  </xsl:template>
```

Figure 3. A piece of the automatically generated XSLT script to generate the HTML pages.

diagrams in a set of different pages with hyperlinks according to the Rigi hierarchical structure. One major benefit of the Visio diagram is the possibility of visualising nested UML packages at any level of details.

The HTML pages are used to browse the architectural model in a textual format. The web interface contains two pages. The left page contains the menu for navigating through the system hierarchy. The user can start from the top level view of the model and navigate the hierarchy by opening and closing the cascading menus. The right page contains the details about the artefacts that have been selected in the left page. The page contains a brief description of the element (e.g. its name and its time) and references to the other elements that have some dependencies with it. Hyperlinks allow the user to navigate through the chain of dependencies.

We have used XSLT to support the conversions of the GXL data to various formats like RSF, Prolog, HTML and DOT. The XSLT scripts for generating the HTML pages are automatically generated by a Tcl script according to the Rigi domain of the graph. Figure 2 shows the XSLT script for converting the GXL file to RSF. Figure 3 shows the automatically generated XSLT script to generate the HTML pages for a particular Rigi domain.

The main features of the environment are: (1) the ability to generate UML nested diagrams in Visio and in HTML with hyperlinks among the different views, (2) the generation of web pages with hyperlinks to support the textual navigation of the architectural documentation, (3) the configuration of the process according to the architectural concepts of the system and (4) the possibility of generating high-level documentation of the system.

One element is that the process and the environment is totally independent from the language and the underlying architecture of the system. It is independent from the types of artefacts or dependencies that we have defined in the first phase.

8 AN EXAMPLE

We demonstrate the approach with a simple example based on the system HAVA. HAVA is a message sequence chart visualiser written in Tcl/Tk and consists of about three thousands lines of code and 230 procedures.

Concept definition

Initially, we discuss with the developer of HAVA the existing design documents and his opinion about the HAVA's architecture. The main computational units of HAVA are Tcl procedures, organised in a set of subsystems with a particular functionality. The developer draws in a diagram his opinion about the current HAVA's architecture. The diagram is visualised in Figure 4 and represents the intended architecture of HAVA where the highest-level elements are shown. In this simple case, we identify three architecturally relevant concepts: the Tcl procedures, the subsystems and the procedure calls.

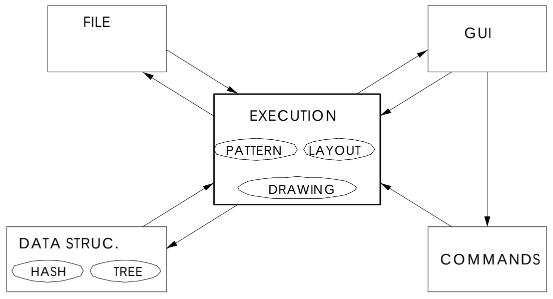

Figure 4. The intended architecture of HAVA.

Documentation Extraction

We use SourceNavigator to parse the source code and to extract the source code model, which represent a low level documentation of the system containing only the Tcl procedures. We point out that in this simple case SourceNavigator is enough to obtain all necessary information. In real cases, we often need to develop our own parsers to extract additional information. The extracted facts are stored in GXL format.

Documentation Abstraction

We use the Nimeta reconstruction environment to create an architectural model of HAVA. With the help of the developer we define a set of rules that create the correct grouping of functions in the subsystems. The grouping rules are expressed as Prolog propositions as we have described in [15]. We use the environment to apply the grouping rules to the source code model and to save the results in a GXL file. Below there is a segment of the resulting GXL file containing also the abstractions:

```
<node id="n_1">
    <attr name="name"> <string>GUI</string> </attr>
    <attr name="type"> <string>System</string> </attr>
    <graph id="g_1" edgeids="true">
          <node id="n_2">
               <attr name="name">
                    <string>GUI_CORE</string>
               </attr>
               <attr name="type">
                     <string>Component</string>
               </attr>
               <graph id="g_2" edgeids="true">
                     <node id="n_3">
                         <attr name="name">
                             <string>gui_areaMessages()</string>
                         </attr>
                         <attr name="type">
                              <string>Function</string>
                          </attr>
                    </node>
                ...
                </graph>
            </node>
            ...
        </graph>
  </node>
```

The *node* elements present the instances of the architectural concepts. The *graph* elements group *node* instances that belong to the same subsystem. GXL allows us to store the documentation elements and their hierarchical positions.

Documentation presentation

For the HAVA case, we have generated the component view of its architecture. The component view shows the main subsystems and the high level dependencies. The Figure 6 shows the component view in UML format. The diagram has been generated in HTML format using Visio. The nested UML packages show only one level of nesting but we can visualised a user-defined level of nesting. Anyway, we do believe that two or three levels of nesting are sufficient for real cases.

We can note that in the reverse engineered view there are many differences from the intended architecture drawn in Figure 4. For example, the “FILE” component depends on the “DATA_STRUCTURES” component, while the developer did not originally want this dependency. In the intended design, the “EXECUTION” component should represent the central element of the system but this is far from the reality.

In Figure 5, the user can navigate the HTML by clicking with the pointer on the UML elements. The navigation allows the user to show more details about the diagram. For example, the Figure 6 shows the details of the subsystem “GUI”.

The Figure 5 shows the component view of the HAVA architecture in HTML format. The left panel shows the top-level view of the HAVA component view. The user can select a particular subsystem to get more details that are visualised in the right pane. The user can also open the cascade menu of a subsystem to show the elements contained in it.

9 OBSERVATIONS

The overall performance of the approach is satisfying. We can store all the reverse engineered documents in the GXL file and we can use XSLT to convert the data to other formats in a very efficient way. Although, we currently generate the Visio diagrams directly from Rigi, XSLT is a candidate for replacing Rigi in this process.

After using the documentation, we find it meets the first three criteria of a flexible documentation mentioned in the Section 1. We provide the information at different abstraction levels and the user is able to move smoothly from one level of abstraction to another, without loosing his current position; the different levels of abstraction are meaningful for the users. We also support the last criterion about the consistency with the code because our process is fully automated and we can re-apply at each new build of the system. Furthermore, the documentation offers various presentation formats for the different types of users. The usage of documentation is enriched.

10 CONCLUSIONS AND FUTURE WORK

In this article, we have described our GXL based approach to generate architectural documentation. The approach consists of two phases: architecture reconstruction and architecture re-documentation. The architecture reconstruction is the process of building architectural model and views; the architecture re-documentation is the process of how to present those views. We demonstrate our approach with a simple example and derived basic observations.

Our approach is flexible and satisfies the essential requirements for the task of architectural re-documentation that is simple, various, meaningful and independent and provides consistency with the source code. This has been achieved by adding links between source code and the documentation.

Our approach takes advantage of GXL, a sub-language of XML, which makes the approach more easy and efficient to generate various presentations. HTML and Visio samples are showed in this paper. Furthermore, we plan to investigate other formats like SVG and VML for the generation of software diagrams.

11 REFERENCES

1. Bowman T., Holt R. C., and Brewster N. V., Linux as a Case Study: Its Extracted Software Architecture, Proc. of the International Conference on Software Engineering (ICSE '99), Los Angeles, May 16-22,1999.

2. C. Riva and J. V. Rodriguez, Combining Static and Dynamic Views for Architecture Reconstruction, *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR 2002)*, IEEE Computer Society Press, 11-13 March 2002 in Budapest, Hungary.

3. Chikofsky E. J. and Cross J. H. II, Reverse Engineering and Design Recovery: A Taxonomy, *IEEE Software*, 7(1): 13-17, 1990

4. Finnigan P.J., Holt R.C., Kalas I., Kerr S., Kontogiannis K., Müller H.A., Mylopoulos J., Perelgut S.G., Stanley M., Wong K., The software bookshelf, IBM Systems Journal, 36(4), October 1997, pp.564-593.

5. Graph eXchange Language, GXL, version 1.0, http://www.gupro.de/GXL

6. Hofmeister C., Nord R.L. and Soni D., Describing Software Architecture with UML, Proc. of the 1st Working IFIP Conference on Software Architecture, Kluwer Academic Publishers, 1999.

7. Kazman R., O'Brien L. and Verhoef C., Architecture Reconstruction Guidelines, Carnegie Mellon University, Software Engineering Institute report number CMU/SEI-2001-TR-026.

8. Kazman R., Tool Support for Architecture Analysis and Design, *Joint Proceedings of the SIGSOFT '96 Workshops (ISAW-2)*, ACM, 1996, pp. 94-97.

9. Kruchten P.B., The 4+1 View Model of architecture, IEEE Software, 12(6):42-50, 1995.

10 Medvidovic N. and Taylor R.N., A Classification and Comparison Framwork for Software Architecture Description languages, *IEEE Transactions on Software Engineering*, Vol.26, No.1, January 2000.

11. Murphy G. C. and Notkin D., Reengineering with Relfextion Models: A Case Study, *IEEE Software*, 1997.

12. Nentwich C., Emmerich W., Finkelstein A. and Zisman A., BOX: Browsing Objects in XML. Software Practice and Experience 30(15):1661-1676. 2000.

13. Ran A., “ARES Conceptual Framework for Software Architecture” in M. Jazayeri, A. Ran, F. van der Linden (eds.), Software Architecture for Product Families Principles and Practice, Addison Wesley, 2000.

14. RedHat Source Navigator, http://sources.redhat.com/sourcenav/

15. Riva C., Architecture Reconstruction in Practice, IFIP Working Conference on Software Architecture (WICSA 2002), August 25-30, Montreal, 2002

16. Riva C., Reverse Architecting: an Industrial Experience Report, Proceedings of the 7th Working Conference on Reverse Engineering (WCRE2000), Brisbane, Australia, 23-25 November, 2000.

17. Riva C., Xu J. and Maccari A., Architecting and Reverse Architecting in UML, Workshop on Describing Software Architecture with UML, International Conference on Software Engineering 2001 (ICSE), Toronto, May 2001.

18. Tilley S. R., Wong K., Storey M.-A. D., and Müller H. A., Programmable reverse engineering, *International Journal of Software Engineering and Knowledge Engineering*, pages 501-520, December 1994. Rigi: a visual tool for understanding legacy systems, University of Victoria, http://www.rigi.csc.uvic.ca/

19. van Deursen A. and Kuipers T., "Building Documentation Generators", *Proceedings of the International Conference on Software Maintenance* (ICSM '99), 1999

20. Visio 2002, http://www.microsoft.com/office/visio/

21. XML, The Extensible Markup Language, W3C Recommendation, 2nd edition 6 October 2002, http://www.w3.org/XML/

23. XSLT, The Extensible Stylesheet Language Tranformation, W3C Recommendation 16 November 1999,http://www.w3.org/TR/xslt

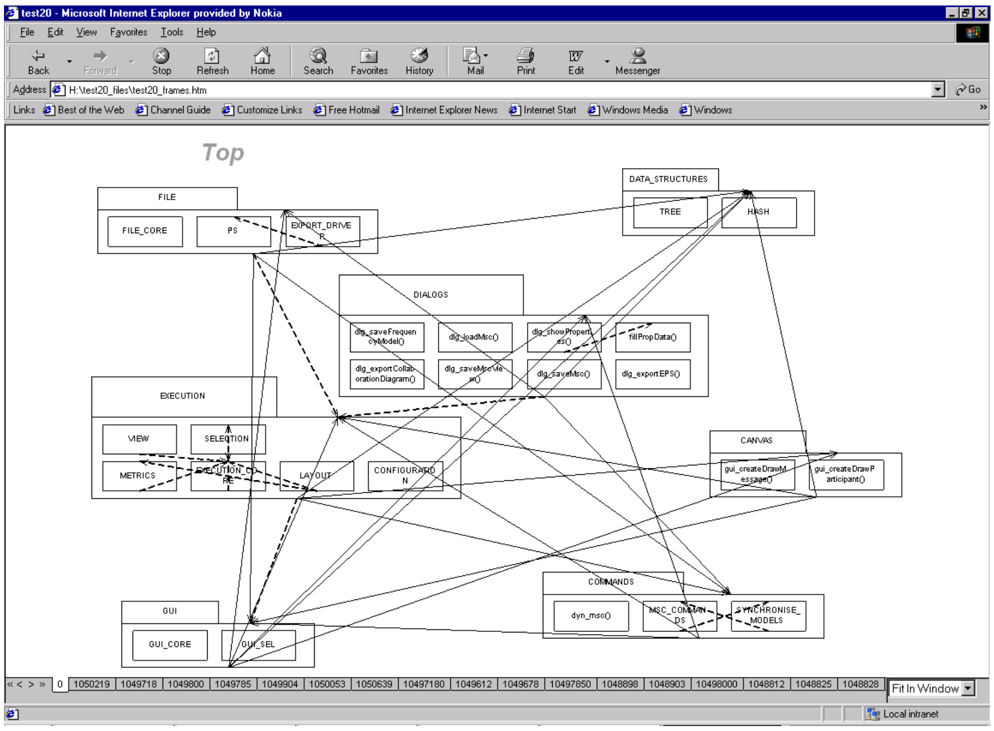

Figure 5. The top level view of HAVA (Visio UML diagrams in HTML).

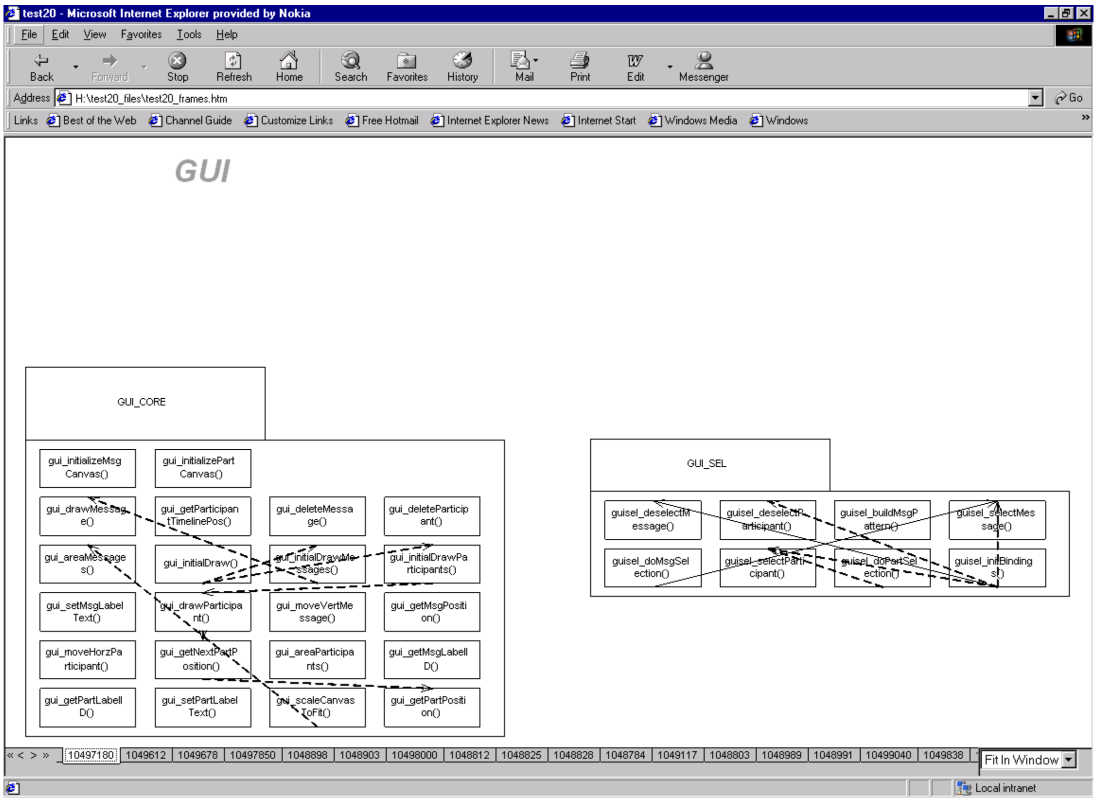

Figure 6. The GUI subsystem of HAVA.