Documenting Software Systems with Views II: An Integrated Approach Based on XML

Jochen Hartmann
BMW Group
Munich, Germany
Jochen.Hartmann@bmw.de

Shihong Huang
Dept. of Computer Science
Univ. of California, Riverside
shihong@cs.ucr.edu

Scott Tilley
Dept. of Computer Science
Univ. of California, Riverside
stilley@cs.ucr.edu

ABSTRACT

Software engineers rely on program documentation as an aid in understanding the functional nature, high-level design, and implementation details of complex applications. Without such documentation, engineers are forced to rely solely on source code. This is a time-consuming and error-prone process, especially when one considers the amount of information assimilation and domain mapping that is required to understand the architecture of a large-scale software system. This paper describes an integrated approach to documenting software systems based on XML. In particular, the paper focuses on the creation and use of specific Document Type Definitions (DTD) that are defined by MSR as a standard for software documentation. MSR is a consortium of several German automotive companies whose goal is to support cooperative development between car manufacturers and their electronic system suppliers. To illustrate the approach, selected aspects of the document creation process for an engine control system are presented.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – Documentation, Enhancement, Restructuring, reverse engineering, and reengineering

General Terms

Documentation, Economics, Management

Keywords

Software documentation, reverse engineering, MSR MEDOC, XML

SIGDOC'01, October 21-24, 2001, Santa Fe, New Mexico, USA.

1. INTRODUCTION

The growing body of legacy software represents significant challenges to organizations relying on complex applications to support their critical day-to-day activities. To properly maintain and evolve an existing application it must first be understood. Gaining such an understanding is complicated by factors such as incremental and crisis-driven maintenance, obsolete programming languages and hardware, and a lack of coherent documentation.

Software engineers rely on program documentation as an aid in understanding the functional nature, high-level design, and implementation details of complex applications. The lack of high-quality documentation is particularly acute for large-scale software systems, since gaining a sufficient understanding of their singular complexities is an extremely challenging task. Without such documentation, engineers are forced to rely solely on source code. This is a time-consuming and error-prone process, especially when one considers the amount of information assimilation and domain mapping that is required to understand the architecture of a large-scale software system.

Reverse engineering is an emerging branch of software engineering that can aid systematic evolution by providing a software engineer with a better understanding of the subject system's current design and overall structure. Reverse engineering is the process of extracting system abstractions and design information from existing software systems for maintenance, reengineering, and evolution. It is seen as an activity that does not change the subject system; it is a process of examination, not a process of alteration [3]. It directly supports the essence of program understanding: identifying artifacts, discovering relationships, and generating abstractions. This process depends on several factors, including one's cognitive abilities and preferences, one's familiarity with the application domain, and the set of support facilities provided by the reverse engineering environment. Reverse

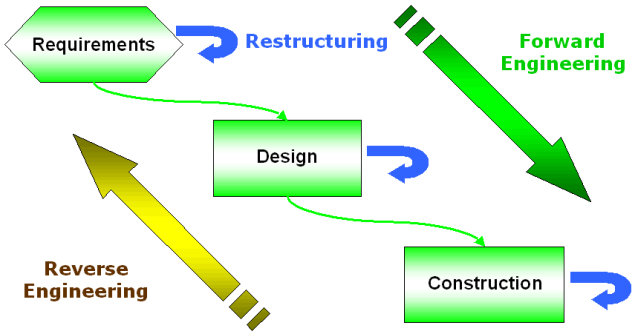

Figure 1: Reverse Engineering in Context

engineering in the context of the overall software lifecycle is shown in Figure 1.

As part of a larger project focused on the documentation of embedded real-time control systems, we have developed an integrated approach to software documentation based on XML [17] that relies in part on reverse engineering. It can be viewed as a sequel to earlier work focused on the same problem [13], but updated to reflect changes in maturing technology and new opportunities presented by industry-wide standards such as XML. In particular, the paper focuses on the use of specific Document Type Definitions (DTDs) that are defined by MSR (Manufacture Supplier Relationship) as a standard for software documentation. MSR is a consortium of several German automotive companies whose goal is to support cooperative development between car manufacturers and their electronic system suppliers [9].

The next section describes some of the shortcomings in current documentation techniques. Section 3 explains the role of XML and the MSR standard in the integrated documentation process. Section 4 highlights some of the benefits of the approach by offering examples taken from a production system. Finally, Section 5 summarizes the paper and briefly discusses future work.

2. SHORTCOMINGS IN CURRENT DOCUMENTATION TECHNIQUES

There are numerous shortcomings in current program documentation techniques. Some of these deficiencies have existed since the earliest days of software development, such as the lack of consistency between the source code and the accompanying documentation. Other deficiencies have only recently emerged as significant issues, such as the difficulty in integrating existing documentation with newly created artifacts for long-lived systems. This section discusses four deficiencies in current documentation techniques that the approach described in Section 3 addresses: low quality, multiple perspectives, document integration, and fundamental requirements.

2.1 Low Quality

High-quality program documentation can greatly aid a software engineer in their tasks by providing multiple complimentary views of the software system [12]. Regrettably, high-quality documentation is rarely considered a priority in a typical development environment. Other engineering tasks, such as design, construction, and testing, always seem to take precedence. This remains true even though the engineers may realize the importance of good documentation as an aid in understanding the system later in its post-deployment stages – when the rationale behind rushed decisions is lost somewhere in the code. Even disciplined organizations that diligently adhere to mature software processes often find it difficult to keep documentation up to date, in part because it is human nature to postpone work with little immediate perceived benefit in comparison to other pressing deadlines.

Indeed, it is almost a software engineering axiom that source code of any quality is accompanied by low-quality documentation. This was true nearly ten years ago, when the first part of the work described in this paper began [12], and it remains true today. As before, one reason for the general lack of quality documentation may be that preparing software documentation is still one of the last activities to take place during program development.

Software documentation seems to be of little interest to the development community. Indeed, recent trends towards light-weight development processes such as Extreme Programming [2] further reduce the emphasis on software documentation as an aid in program understanding. While this technique may work for new development (the community has not yet reached a consensus on this), it certainly won't help engineers struggling to understand existing software systems that are undergoing evolution.

2.2 Multiple Perspectives

Software documentation has many different audiences. Different levels of documentation are required for the casual user of the program, for the developer familiar with the code, for the calibrator unfamiliar with the performance characteristics of the program, for testers and technical writers trying to understand the program's functionality, and for project management personnel looking for the gestalt – a "big picture" view of the system's architecture and history [13].

In addition to having many different users, software documentation also has many different authors. Different developers write in their own style, using their own way to describe their ideas. Engineers often work at different levels of abstraction, starting with the specification of the system, implementing the required functionality, and ending with the calibration and testing of the system. All these different engineering tasks have their own view and understanding of the same artifact: the system they are working on. The goal of each documentation process is to combine the views of the authors to generate views suitable for multiple perspectives of the diverse users.

For the development of an automotive control system, engineers and specialists from several disciplines work together. All these team members write documents with there own point of view and their own style. This is understandable, since they are writing documents from different backgrounds and experiences, and with different goals. However, they are describing the same situation. To understand the whole system, including its operation, it is necessary to take all available documentation into consideration.

2.3 Document Integration

To properly perform maintenance on an intricate application, such as an automobile's embedded real-time control system, the engineer must examine the source code of several different components. Source code represents the solution part of the engineering equation; a description of the problem itself is often missing, incomplete, or difficult to use because it is usually in an unstructured format. To understand a system, both parts are required: the problem domain and the solution domain. These are two different worlds, the world of user-oriented requirements and the descriptions of the system environment, and the world of the engineer-oriented solution, realized as the system itself. It is no longer possible to understand a complex, large-scale software system solely based on source code analysis.

To improve understanding, the engineer must also use secondary documentation sources, such as configuration management data and trace events from simulations. Tool support for analyzing the system's source code is available, but the tool often doesn't provide the data in a format that the engineers need. More importantly, the data provided by the tool is difficult to integrate into the rest of the development process, thereby limiting its usefulness in day-to-day tasks within a development environment.

The problem is further exacerbated when one considers that to properly understand the software system and its operational context, the engineer is forced to understand

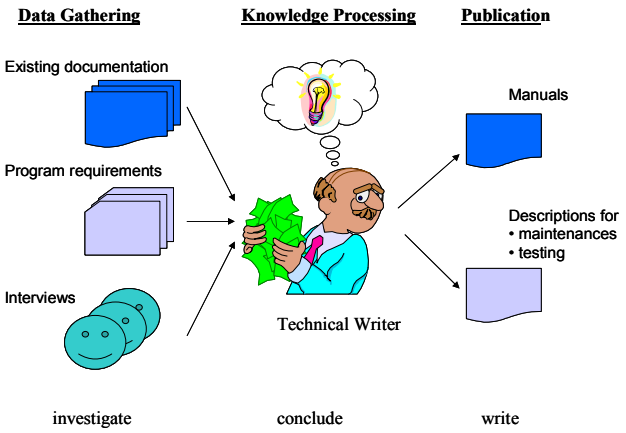

Figure 2: Manual Document Processing

more than just source code. The software engineer may have to examine parts of the overall system (including hardware interfaces and subsystems) and selected aspects of the overall automotive assembly. Manually gaining a sufficient understanding of such a complex entity can be extremely challenging for even the most adept engineer. Nevertheless, the engineer must make judicious use of this ancillary information, because only a small part of the information necessary to understand a software system is located in the code. If this information is kept in different sets of documentation, integrating these disparate sources of information can be very difficult. However, if it is done properly, document integration can greatly aid understanding; it addresses some aspects of the multiple perspectives problem discussed in Section 2.2.

The problem is similar for the technical writer responsible for creating user documentation (such as manuals) and for creating internal documentation used by other departments (such as testing procedures). As illustrated in Figure 2, the communication professional must gather information from multiple sources, such as existing product documentation, original program requirements and specifications, and even interviews of developers. This type of documentation integration is just as difficult as that facing the software engineer.

2.4 Fundamental Requirements

There are several highly-specialized conferences that focus on the constructions of tools to aid program understanding by creating ancillary documentation, providing graphical views of the software system, and generally attempting to augment the knowledge hidden in the source code with secondary structures. However, these tools remain relatively underused. Perhaps one reason for their slow adoption rate is that the type of information these tools

produce is not the type of information that the software engineer needs to complete their task.

As stated in Section 2.1, high-quality program documentation can greatly aid the engineer by providing multiple and complimentary views of the software system. However, the truth is that no one really knows what types of documentation are truly useful [10]. This highlights an underlying problem with current documentation techniques: if no one understands what is needed, it should come as no surprise that tools which produce this type of documentation are rarely used by real-world software engineers. This situation raises several questions about the fundamental requirements of program documentation:

- What types of documentation does a software engineer need? What formats should it take? For example, inline or linked textual commentary? Graphical views? Multimedia?
- Who should produce the program documentation? The engineers themselves? The testers? The technical writers? Who should maintain the documentation once it is produced?
- When should the program documentation be produced? During development? After the product has been deployed? There is a school of thought that "just in time" documentation might prove more useful than providing all the documentation at once. While the typical problem is a lack of high-quality documentation, there can also be a problem of too much documentation. Or, more accurately stated, too much documentation not in the correct format for the task at hand.
- How should the program documentation be written? Are there particular styles of writing that make software documentation more understandable? Is a minimalist approach better? Might a "wizard"-like agent be used to guide personalized content creation?

Rather than attempting to directly address these fundamental issues, we have side-stepped the questions somewhat by focusing on an integrated approach to documentation that relies on standard techniques and standard formats; we let community as a whole deliberate and let the individual user decide what constitutes "good" documentation for their particular task.

3. PROGRAM DOCUMENTATION IN XML

One way of producing accurate documentation for an existing software system is through reverse engineering. Previous attempts to automatically create program documentation have usually relied on proprietary tools. For example, in [13] an approach is described that used a prototype reverse engineering environment to create views of the underlying software system to aid in its understanding. This approach, and the tools developed to support the approach, proved quite successful in earlier case studies (e.g., [20]). However, transitioning such a research tool into a commercial development environment is challenging, in part because the tools relies on proprietary data formats, beta-level applications, and undocumented interfaces. Commercial enterprises are understandably reluctant to adopt such a toolset, even if the underlying methodology is sound.

This paper describes an updated approach to program documentation that had its genesis in earlier work focused on the same problem [13]. The new approach reflects changes in maturing technology and new opportunities presented by industry-wide standards to create a more complete and transitionable solution. This section provides an brief overview of the underlying technology, XML, the specific Document Type Definitions (DTD) defined by MSR as a standard for software documentation, and the process used to create MSR documents.

3.1 XML

XML (eXtensible Markup Language) is subset of the Standard Generalize Markup Language (SGML), an ISO standard for defining and using document formats. XML is a markup language for documents containing structured information, capturing both the content and meaning of the document. The XML specification defines a standard way to add markup to documents.

Syntactically, XML superficially resembles HTML 0[16]. However, three important respects of XML make it more useful for program documentation than plain HTML: extensibility, structure, and validation. XML is extensible in that it allows users to specify their own tags or attributes in order to parameterize or otherwise semantically qualify their data. When using XML in program documentation, the software engineer or technical writer can define their own tags, such as <TASK>, <FILE>, <FUNCTION>, <VARIABLE>, <CONSTANT>, etc. Such application- or domain-specific tags provide implicit semantics to the document and enable the use of automated processing by secondary tools. For example, common tools can be used to search through a document collection for a set of related functions.

The hierarchical nature of XML documents provides useful structuring mechanisms for program artifacts. For example, XML constructs can be used to represent both database

schemas and database elements that capture important source-level information from the software program. The same structuring mechanisms can be used to integrate documents from different sources, partially alleviating the problems outlined in Section 2.3.

To ensure consistency in the data captured in a document, XML allows consuming applications to check data for structural validity. Users can check if transmitted data is valid and adheres to the agreed-upon DTD. The validation is an important feature to make sure the data is exchangeable among different software systems. This type of constraint checking helps to maintain consistency and quality across multiple documents, ensuring that each user sees and interprets the same document in the same fundamental way. However, it does not preclude transformations of the data via XSLT [18] to satisfy the multiple perspectives problem discussed in Section 2.2.

XML has been used for a wide variety of documentation-related applications. For example, the U.S. Congress uses XML and several application-specific DTDs to make legislation available to the general public [15]. The main programs in Microsoft Office XP (e.g., Word, PowerPoint, and Access) can store their documents natively in XML format [7].

In our case, using XML as a standard format for program documentation makes good sense. XML documents can be processed by commonly-available third-party tools (e.g., editors, browsers, parsers), alleviating the need for us to construct special-purpose tools that would inevitably less robust (at least in the short term). XML documents can also be stored, queried, and updated in a relational database, either in native format or via automated transformations between the textual XML and the internal database representation. The DTD used for program documentation in our automotive-related project is MSR.

3.2 MSR

The modern automotive industry is characterized by distributed engineering processes. The asynchronous and geographically dispersed nature of these processes makes it necessary that all participants exchange information using the same representation. Like humans speaking to one another using the same language, the different participants of an engineering process have to define there own "language" to define the meaning of the words and the content. Technologies such as XML offer the opportunity to describe the information that is necessary to work together in a distributed development in a coherent and contributory manner. For the automotive industry, that standard is MSR.

MSR is a consortium of several German automotive companies whose goal is to support cooperative development between car manufacturers and their electronic system suppliers. The companies participating in the MSR consortium include BMW AG, DaimlerChrysler AG, Porsche AG, Robert Bosch GmbH, and Siemens Automotive.

By using a standard representation such as MSR, a shorter development process is possible through the improved flow of information and exchange of engineering data. The defined method for specification and documentation allows the reduction of errors in the development process between manufacture and supplier. This results in an improved quality and lower costs.

Within the MSR consortium there are several projects established to describe the different parts of the development process. For example, MSR MEDOC (MSR Engineering DOCumentation) is used for documentation, and MSR MESA/MERLAN is used for development, simulation, analysis and verification. The role of MSR is not to mandate any standardization of the system or the features that are described with MEDOC. Instead, MEDOC supports the use of both (inter)national standards and in-house conventions, as well as non-standard usages, for the description of systems of data relevant to the documentation of development process.

The MSR MEDOC standard is divided into several sub-standards:

- MSR-SW (SoftWare) to describe the elements of the software (e.g., variables, functions, relationships)
- MSR-REP (REPort) for reports and for documentation of earlier phase of the development process (e.g., functional requirements)
- MSR-SYS (SYStem) for the description of the hardware (sensors, ECU …).
- MSR-NET (NETwork) for the documentation of the network(s) within a car

The documentation process described in the next section focuses on the MSR-SW documents.

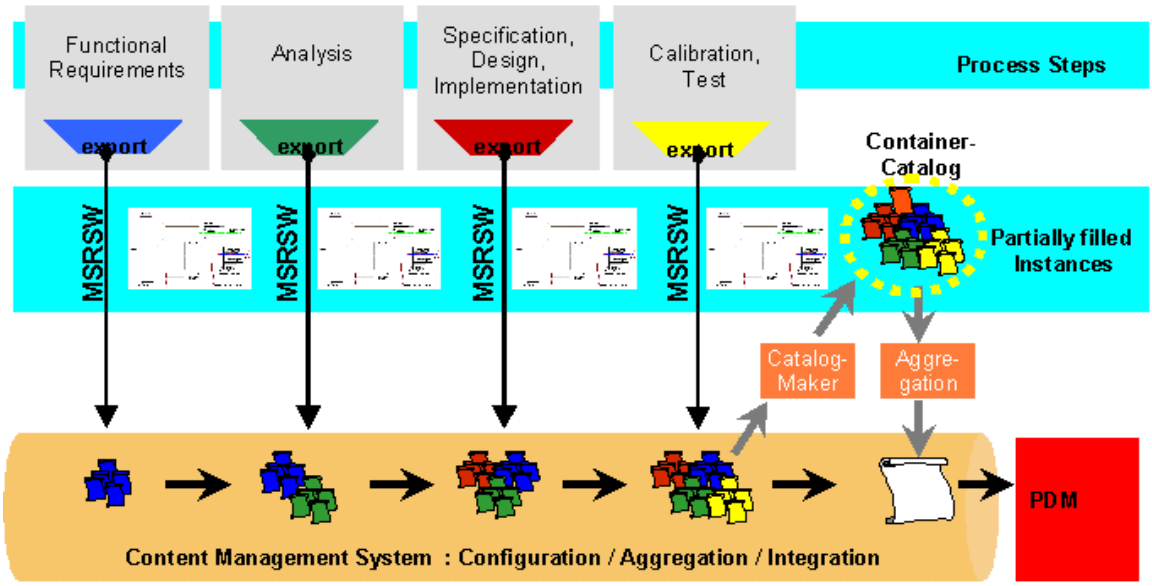

Figure 3: Documentation Process (image taken from [5])

3.3 The Documentation Process

The integrated process used to create and maintain high-quality documentation relies on maintaining consistency between the program documentation and the underlying software. Taken together, the software and the documentation represent a system-specific configuration that is stored in a configuration management system. An illustration of the documentation process is shown in Figure 3; this paper focuses on the analysis portion.

As illustrated in Figure 4, the integrated approach to program documentation can be broken down into three distinct activities:

1. Data Gathering
2. Knowledge Processing
3. Publication

Data gathering is concerned with extracting information from the source code, from existing documentation, and from users. The source code is processed using a commercial parsing system and the relevant artifacts and relationships are extracted. The data is stored in a commercial relational database system.

Existing documents are also used as input to the data gathering process. Writing documentation means collecting information, making investigations, combining this information with other sources, inferring new facts, understanding the whole, and generating updated documentation. Therefore, creating new documentation can be interpreted as the reengineering of existing documentation. Similarly, redocumentation can be viewed as the same process as regular documentation, only the data sources are different.

The third form of document data gathering is incorporating user input. Senior developers often carry implicit knowledge concerning the system, such as why particular design decisions were made. By making this knowledge explicit and storing it in the same repository as the other two sources of data (source code and existing documents), a "Sammeltopf" is created that captures the essential documentation building blocks that are used in the knowledge processing phase.

Knowledge processing refers to a semi-automated manipulation of the underlying database, with the goal of uncovering important information hidden in the gathered data. This activity is eminently suited to automated techniques such as data mining and relationship analysis. The goal is not to show the user interrelation, which can be seen by examining the code. Instead, knowledge processing offers the possibility to show correlation, something that is normally difficult to see.

For example, timing information related to reading and writing of variables by functions is of critical importance

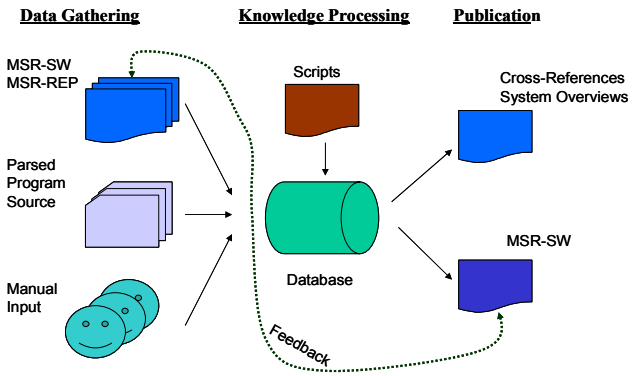

Figure 4: Semi-Automated Document Processing

for real-time systems. By using the data gathered by the parser from the source code, combined with ancillary documentation related to system design and tasking constraints, the actual periodicity of variable access can be determined and potential problems identified.

The publication activity results in the creation of both textual and graphical forms of documentation. For example, cross-reference information and system overviews that are directly related to the source code and system architecture are often presented in graphical format (examples are discussed in Section 4.2). While most tools for program understanding are based on the visualization of graphs, textual representations are also useful. The integrated approach can produce such documentation in standardized XML format, as discussed above.

Combined with traditional documentation techniques, the new integrated documentation system can be an effective facility to provide engineers with much of the relevant information required to perform disciplined evolution of the software system. Some of the benefits of this approach are elaborated in the next section.

4. BENEFITS OF AN INTEGRATED DOCUMENTATION APPROACH

The integrated approach to program documentation outlined in Section 3.3 offers several benefits over the traditional techniques and directly addresses some of the shortcomings outlined in Section 2. This section outlines three such benefits: high quality, multiple perspectives, and document integration.

4.1 High Quality

In Section 2.1 it was stated that high-quality program documentation can greatly aid software engineers in their

```
call($db,"f3",1,"in");
call($db,"f3",1,"out",0);
access($db,"f2","constAccess","r",0);
access($db,"f2","varAccess","r",0);
access($db,"f2","varAccess","rw",0);
access($db,"f2","varAccess","w",0);
access($db,"f3","varAccess","r",0);
access($db,"f3","varAccess","rw",0);
access($db,"f3","varAccess","w",0);
```

Figure 5: Programmatic Document Creation

tasks. However, traditional approaches to program documentation often result in low-quality documents, in part because of the difficulty in manually keeping documentation synchronized with source code. The integrated approach to program documentation addresses this problem by creating high-quality documentation using a semi-automatic process.

By automating those parts of the program documentation process that are repetitive and error-prone, the resultant documents are guaranteed to be an accurate reflection of the source code. Since the process is automatic, it can be executed at any time, for example during regular compilation or when the rest of the documentation is post-processed, thereby solving the problem of program documentation being out of date.

All three parts of the process (data gathering, knowledge processing, and publication) are automated, but the approach provides for human intervention and guidance where desired. By using XML as the underlying representation format, the documentation is both human-readable and machine-processable. Moreover, the output of the publication process can be fed back into the next

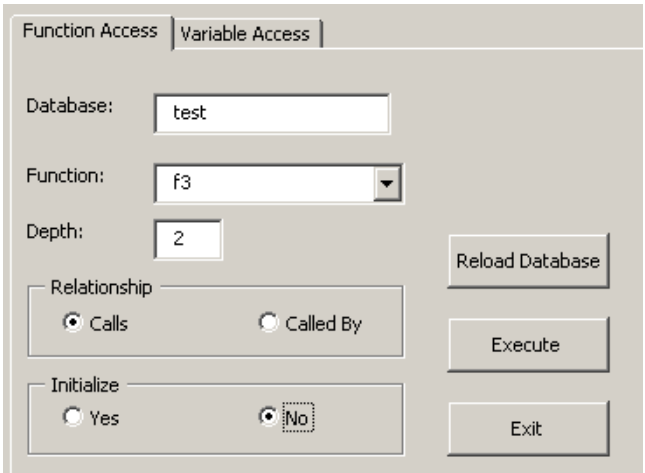

Figure 6: Graphical Document Creation

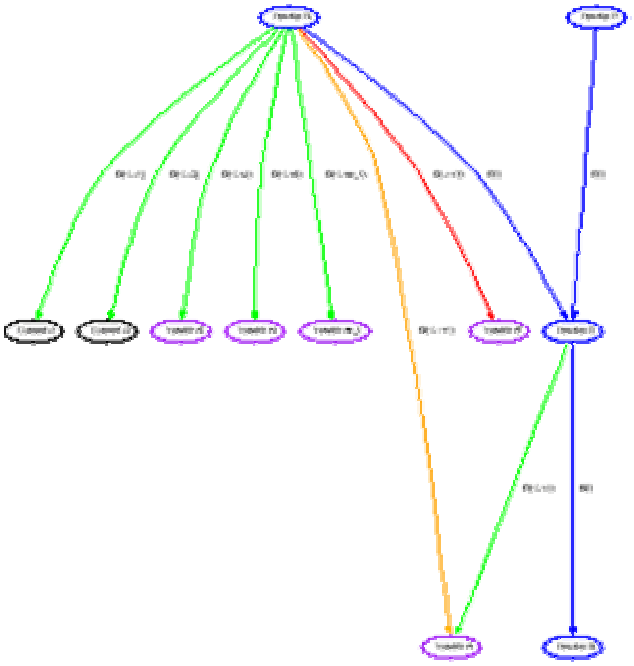

Figure 7: Graphical Document View 1

iteration of the data gathering phase to refine the information contained in the repository.

4.2 Multiple Perspectives

As stated in Section 2.2, software documentation has many different audiences. Different levels of documentation are required for the casual user of the program. Some people prefer to see documentation in a graphical format, while others prefer to see documentation in a textual format [14]. Since the ultimate goal of this type of documentation is to aid program understanding, both perspectives should be supported.

The integrated approach provides multiple, user-selectable (and user-defined) views of the underlying information through both programmatic and graphical interfaces. For example, the Perl code fragment shown in Figure 5 illustrates the programmatic interface to the semi-automated documentation process, while Figure 6 is one of the graphical widgets available to the software engineer. Irrespective of the method of invocation, the result of running this code or query is to populate the underlying database with function call and constant/variable access information for several functions.

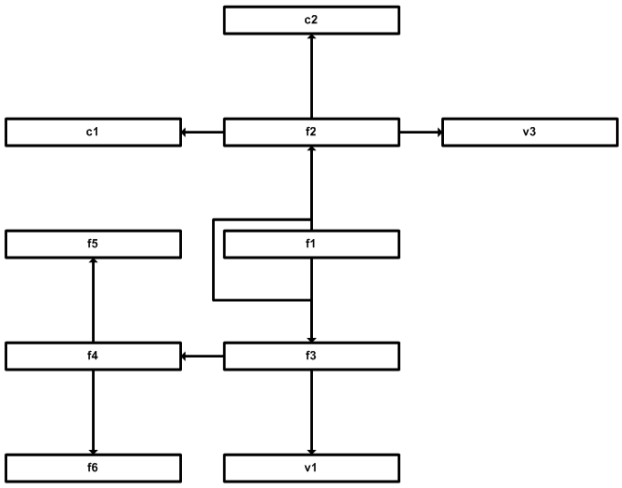

Figure 8: Graphical Document View 2

This information can then be used and viewed from multiple perspectives. For example, Figure 7 illustrates the result of executing the code fragment shown in Figure 5 against data gathering from an engine control system using the research-oriented graph visualization program "dot" [1]. Figure 8 shows the same information, but this time using the common Microsoft Office drawing application Visio [8]. Similarly, Figure 9 shows an alternate perspective of the same data, but in textual XML format, processed by the XMetal program [11]. The same XML text is show in Figure 10 using the industry-standard Web browser Internet Explorer [6].

There is nothing particularly novel about these views, other than the fact that they are automatically generated and can capture multiple perspectives of the underlying software system on demand. The focus of the integrated document process is not the creation of new visualization techniques; rather, the focus is on a holistic approach to integrating existing applications into the process.

4.3 Document Integration

The third benefit associated with an integrated documentation process based on XML is the ease with which other sources of information can be incorporated into the document creation procedures. By using standard representations, such as XML and the MSR DTDs, as well as commonly-available and popular tools for data gathering and document viewing, many of the adoption problems encountered in previous attempts at automated document creation are alleviated.

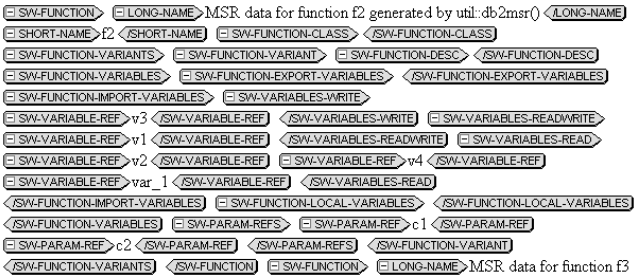

Figure 9: Textual Document View 1

If views are required which reflect information from more than one engineering tool, then either some form of control and data integration among the tools is required, or the documents must be assembled manually. This gap between the underlying engineering repository and the document views can be closed by introducing a middle layer, the document base [19]. This layer is built upon a standard data format – the MSR documents – that can be used to generate all the required document views in the presentation layer. Documentation based on XML offers the potential for establishing a glue layer in complex engineering processes.

The use of standards means that other documents related to the entire system – not just the software aspects – can be integrated into the process. For example, the engine control software of an automobile has dependencies on hardware components as well as other control units that it communicates with using local area networks. Considered as a coarser-grained component, this in turn has hierarchical relationships of various types with other components in the overall system. By using the same documentation approach for all such artifacts in the entire automobile, a more complete picture of the intricate relationships within the system emerges.

The use of standards also means that third-party tools can be integrated into the infrastructure provided. For example, a common design tool for real-time systems in the automotive industry is ASCET SD [4]. By integrating the documents both used and produced by this tool into the documentation process, a better understanding of the overall software system can be gained.

5. SUMMARY

High-quality program documentation plays an important role in the understanding of complex systems. Unfortunately, such documentation is in short supply for most software-intensive systems, forcing engineers to rely on the analysis of low-level source code to recreate high-level understanding. When one considers the amount of information assimilation and domain mapping that is

```
- <SW-FUNCTION>
    <LONG-NAME>MSR data for function f2 generated by util::db2msr()</LONG-NAME>
    <SHORT-NAME>f2</SHORT-NAME>
    <SW-FUNCTION-CLASS />
  - <SW-FUNCTION-VARIANTS>
    - <SW-FUNCTION-VARIANT>
        <SW-FUNCTION-DESC />
      - <SW-FUNCTION-VARIABLES>
          <SW-FUNCTION-EXPORT-VARIABLES />
        - <SW-FUNCTION-IMPORT-VARIABLES>
          - <SW-VARIABLES-WRITE>
              <SW-VARIABLE-REF SW-VARIABLE="v3">v3</SW-VARIABLE-REF>
            </SW-VARIABLES-WRITE>
          - <SW-VARIABLES-READWRITE>
              <SW-VARIABLE-REF SW-VARIABLE="v1">v1</SW-VARIABLE-REF>
            </SW-VARIABLES-READWRITE>
          - <SW-VARIABLES-READ>
              <SW-VARIABLE-REF SW-VARIABLE="v2">v2</SW-VARIABLE-REF>
              <SW-VARIABLE-REF SW-VARIABLE="v4">v4</SW-VARIABLE-REF>
              <SW-VARIABLE-REF SW-VARIABLE="var-1">var_1</SW-VARIABLE-
                REF>
            </SW-VARIABLES-READ>
          </SW-FUNCTION-IMPORT-VARIABLES>
          <SW-FUNCTION-LOCAL-VARIABLES />
        </SW-FUNCTION-VARIABLES>
      - <SW-PARAM-REFS>
          <SW-PARAM-REF>c1</SW-PARAM-REF>
          <SW-PARAM-REF>c2</SW-PARAM-REF>
        </SW-PARAM-REFS>
      </SW-FUNCTION-VARIANT>
    </SW-FUNCTION-VARIANTS>
  </SW-FUNCTION>
```

Figure 10: Textual Document View 2

required in this process, the limitations and shortcomings of such an approach become apparent.

This paper described an integrated approach to documenting software systems based on standardized representations such as XML and commonly-available tools. The specific usage scenario is the documentation process for automotive systems. However, the approach is sufficiently general that it could be applied in other application domains.

There are many possible avenues for future work in this area. For example, an even better understanding of the software system could be gained by providing direct links between the source code and the requirements engineering documents (where available). There are commercial systems for managing requirements documents, which would be integrated into the broader documentation process, to the significant advantage to all team members.

To fully exploit the advantages of standards-based documentation, some of the facilities provided by XML could be better leveraged. For example, augmenting the MSR standard with company- or project-specific tags in a systematic manner would make the approach more extensible and hence make software engineers more receptive to its adoption.

Although the fundamental questions raised in Section 2.4 regarding the underlying requirements of program documentation remain, hopefully the approach described in this paper will go some way towards a solution.

REFERENCES

[1] AT&T Research. "The Dot Graph Visualization Program." Online at www.graphviz.org.

[2] Beck, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[3] Chikofsy, E.; and Cross, J. "Reverse Engineering and Design Recovery: A Taxonomy." IEEE Software 7(1):13-17, January 1990.

[4] ETAS GmbH. "ASCET SD." Online at www.etas.de.

[5] Manger, G. "A Generic Algorithm for Merging XML/SGML Instances." XML Europe 2001.

[6] Microsoft Corp. "Internet Explorer." Online at www.microsoft.com/windows/ie.

[7] Microsoft Corp. "Office XP Professional." Online at www.microsoft.com/office.

[8] Microsoft Corp. "Visio: The Office Business Diagramming Solution." Online at www.microsoft.com/office/visio.

[9] MSR MEDOC. Online at www.msr-wg.de/medoc.html.

[10] Smith, D.; Thomas, B.; and Tilley, S. "Documentation for Software Engineers: What is Needed to Aid System Understanding?" *Proceedings of the 19th International Conference on System Documentation* (SIGDOC 2001: October 21-24, 2001; Santa Fe, NM). New York, NY: ACM Press, 2001.

[11] SoftQuad Corp. "XMetal 2: The Premier XML Content Enabler." Online at www.softquad.com.

[12] Tilley, S. and Müller, H. "INFO: A Simple Document Annotation Facility." *Proceedings of the 9th Annual International Conference on System Documentation (*SIGDOC'91: October 10-12 1991; Chicago, IL), pp. 30-36. New York, NY: ACM Press, 1991.

[13] Tilley, S.; Müller, H.; and Orgun, M. "Documenting Software Systems with Views." *Proceedings of the 10th Annual International Conference on System Documentation* (SIGDOC'92: October 13-16 1992; Ottawa, Canada;), pp. 211-219. New York, NY: ACM Press, 1992.

[14] Tilley, S.; Whitney M.; Müller H. and Storey M.-A. "Personalized Information Structures", Proceedings of the 11th International Conference on System Documentation (SIGDOC'93: October 5-8, 1993; Waterloo, Canada), pp 325-337. New York, NY: ACM Press, 1993.

[15] United States Congress. "XML and Legislative Documents." Online at xml.house.gov.

[16] W3C. HTML. Online at www.w3.org/MarkUp.

[17] W3C. XML. Online at www.w3.org/XML.

[18] W3C. XSLT. Online at www.w3.org/Style/XSL.

[19] Weichel, B. "Strategies for Implementing SGML/XML as a Glue Layer in Engineering Process." SGML/XML Europe 1998.

[20] Wong, K.; Tilley, S.; Müller, H.; and Storey, M.-A. "Structural Redocumentation: A Case Study," *IEEE Software* 12(1):46-54, January 1995.