Identification of Software System Components Using Semantic Models and Graph Slicing

Mark McKenna  Jacob Slonim  Michael McAllister  Kelly Lyons

July 31, 2007

Abstract

In this paper, we present an implementation of part of a process for automatically decomposing a legacy software system into a loosely coupled Service Oriented Architecture. We present the Relationship Modelling Framework (RMF), which, when coupled with the appropriate components, can generate and maintain a semantic model of a software system, showing at various levels of abstraction the elements that make up a software system, including (but not limited to) source code entities, database tables, and configuration resources, and the relationships between these elements. We present a technique called graph slicing, derived from program slicing, which can be used to identify the dependencies of selected elements.

IBM independently verifies the performance of the model generation process, and the accuracy of the graph slicing process, using a prototypical implementation of RMF. The validation is performed against a real-world software suite produced by IBM.

1 Introduction

Software systems evolve over time [8]. Needs and expectations change, motivating the growth and modification of the system; as the system changes, it can become more complex. Increases in complexity motivate refactoring and reengineering to simplify the system and facilitate future change.

The structure of a software system "...comprise[s] software elements, the externally visible properties of those elements, and the relationships among them." [1]. A more structured system may comprise more kinds of elements and relationships, or more restrictions and requirements on those relationships.

The structure of a software system is not always obvious to the engineers of the system. Aspects of a system’s structure can be spread across multiple documents, or visible in some documents (like high level design documents) and not in others (such as source or object code). Often, aspects of the structure of a software system are only visible when several different kinds of document are considered together (for example, source code documents, library files, and a database catalog may all be required to understand how a software system interacts with a database repository). Structure which is less obvious is more difficult to understand and maintain.

We hypothesize that a semantic model of a software system comprised of many different kinds of document can be generated, and that such a model can be automatically processed to provide insight into the structure of the modelled system for the purpose of software decomposition. In this paper we demonstrate that a semantic model can provide insight into the structure of the modelled system, and believe that the application of this insight to software decomposition follows based on the completeness and fidelity of the model.

In this paper we introduce the idea of graph slicing, an application of a graph operation which is analogous to the existing notion of program slicing, to allow for ‘slicing’ of entire software systems.

We develop the Relationship Modelling Framework (RMF), to generate and maintain semantic models of software systems based on the documents which describe the system. We implement a prototype of RMF, which we use to generate a semantic model of a large legacy software system based on the J2EE specification. We perform graph slicing on this semantic model to extract a complete dependency tree for a chosen component of the software system. We analyse the performance of the graph generation process, and the accuracy of the resulting dependency trees, and draw conclusions about the usefulness of this method in support of software decomposition.

The contributions of this work are the Relation-
ship Modelling Framework, the graph slicing concept, and the application of these to the identification of software system components for the purposes of software decomposition.

In section 2 we describe the relevant background literature surrounding our research, including program slicing, semantic modelling and the Service Oriented Architecture (SOA). In section 3 we describe the methodology used to develop our solution, including a process for performing software decomposition. In section 4 we describe the graph slicing procedure, and the RMF architecture. In section 5 we describe the method we used to validate our contributions. In section 6 we describe the results of our validation. In section 7 we draw conclusions regarding the accuracy and utility of RMF and the graph slicing procedure. Finally in section 8 we discuss future work.

2 Background and Related Work

This section describes other tools and techniques that exist for supporting the reengineering of software systems. We also describe semantic modelling, as the technology we use to represent the structure of a software system, and the Service Oriented Architecture, as a practical target architecture for our decomposition process.

2.1 Software Reengineering

Software reverse engineering is the process of deriving elements of a software system’s design through analysis of its structure and behaviour. Software reengineering is a process of iteratively reverse engineering and forward engineering (often simply ‘engineering’) through which the structure and behaviour of an existing software system is modified. One specific type of software reengineering is software decomposition.

2.1.1 Software Decomposition

Software decomposition is an important element of software reengineering [9, 10, 12]. Schmitt defines it as “The process of breaking the description of a system down into small components…”[16]. The goal of software decomposition is to produce components which maximize intra-component cohesion, and minimize inter-component coupling.

One of the primary challenges when decomposing an existing software system is determining along what boundaries to perform the decomposition to achieve the best cohesion and coupling characteristics. A number of techniques [2, 9, 19] have been described for aiding in decomposition. In our research, we deal with one technique in particular, called program slicing.

Program slicing, introduced by Weiser [19], is a technique that produces a program slice, a minimal subset of a program which implements a selected portion of that program’s behaviour. Some kinds of program slice are derived starting from a selection of variables, program statements, and other functional components of a program. The most common slices retain the portion of the code that affects the value of a set of variables, or the outcome of the execution of a set of program statements. In procedural languages, such slices can be computed using the control- and data-flow dependency information about the program.

Variations on program slicing have been described for more granular program structures such as code modules [3], and in an interactive editor-supported fashion [14]. Program slicing has been applied to software development problems that include testing, reuse, knowledge discovery, maintenance and evaluation.

2.1.2 Reengineering Tools

Several tools and techniques exist for aiding the software reengineering process. Unravel [11] is a prototypical program slicing project, designed to implement program slicing on the ANSI C programming language.

Rigi [13] uses a graph-based representation of the code components of a software system to facilitate “semi-automatic” reverse engineering of the system. It automatically builds a low-level model of the system code, then allows the system analyst to construct logical views over the low-level model, creating a semantic model of the system at a somewhat higher level of abstraction. Rigi focuses on modelling system structure evident from the source and/or compiled code objects of the system, producing abstractions as an interactive process.

CodeSurfer [4] is a software system which enables the browsing of system source code at a low level. It “…understands pointers, indirect function calls, and whole-program effects.” [5] CodeSurfer implements many fundamental analyses of source code, including call graphs and control- and data-flow analyses. An integrated visualization tool presents several predefined views of the system model.
2.2 Semantic Modelling

A semantic model [15] abstractly represents the structure or behaviour of a system. It is a very general model which can represent a wide variety of concepts, and the relationships between those concepts. One important capability of a semantic model is the representation of meta-information (information about other information) along with the information it describes. This allows users of the model to reason about the information, deriving new information from it.

One place where semantic models are used extensively is in the Semantic Web [18]. The Semantic Web is an extension of the World Wide Web which exposes web content in machine-readable form, in addition to natural language form. The Semantic Web is the notion of a world-wide hyper-linked network in which the data are individual concepts and the hyperlinks represent the relationships between those concepts. This is in contrast to the ‘legacy’ or existing World Wide Web, a hyper-linked network in which the data are documents and the hyperlinks represent functional associations between those documents.

The Resource Description Framework (RDF) [7] is a language, specified by the World Wide Web Consortium (W3C) [18], for representing a semantic model as a directed, vertex- and edge-labelled multigraph. This representation enables the application of many existing graph algorithms and techniques to the semantic model. It is also a conceptually simple model, and therefore information represented in an RDF graph may be easier to comprehend than information in other, more syntactically complex representations.

In an RDF graph, vertices represent abstract or concrete entities, and edges represent relationships between those entities. Vertices are labelled with a unique identifier which identifies the concept that the vertex represents. Edges are labelled with a string which indicates the type of the relationship it represents. This string is not unique among all edges; however, two edges with an equivalent label are taken to represent the same kind of relationship. As implied by the multigraph designation, two vertices may be connected by a directed edge more than once; however this is limited to distinctly labelled edges.

Figure 1 shows an RDF graph. Statements are shown as edges connecting two vertices, moving from the subject to the object and labelled with the predicate’s identifier. Vertices represent resources, and are labelled with their own identifiers.

2.3 Service Oriented Architecture

Service Oriented Architecture (SOA) is “A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains.”[6] It is an architectural style, one of whose goals is to achieve loose coupling among software agents. An agent is any SOA-capable software system. Agents fall into two overlapping subclasses: services and clients. Services are agents which provide functionality for consumption by clients; clients are agents which use the functionality provided by services. Agents intercommunicate via a message passing mechanism, which enables them to communicate across programming languages, platforms and over networks.

3 Methodology

We identified an experimental methodology through which we could develop a system and technique for automating a portion of the software decomposition process. We refer to the software system being decomposed as the ‘legacy system’, and the system being produced through decomposition as the ‘target system’.

IBM, our industry partner, is interested in the development of technology which supports the reengineering of large and complex software systems. In particular, IBM is interested in decomposing these systems into loosely coupled structures. We remain in contact with IBM personnel throughout the course of this research, to gain insight into the nature of the problem. In order to ensure that our approach is practical, we have IBM independently validate the technology developed from this research.

We first identified a general process which can be used to decompose a legacy software system into a set of services in an SOA. We chose SOA as the target architecture because of its loosely coupled nature. This process is described in figure 2. This result-driven
1. Identify a set of desired services; call these the primary services of the target system.

2. Create a model of the system to be decomposed, showing system elements and interdependencies between those elements.

3. For each primary service identified:
   (a) Identify the model elements which represent the system elements required to enable the functionality of that service.
   (b) Compute the ‘support set’ of the system elements: all system elements on which the identified system elements depend, either directly or transitively.

4. Compute the intersections of the support sets for each primary service; this indicates functionality which is used to more than one primary service.

5. Partition the model to maximize intra-service cohesion, and minimize inter-service coupling. This may involve the creation of secondary services which support the primary services.

6. Implement the identified services using the code identified in the legacy system.

Figure 2: Process for result-driven decomposition of a legacy system into a set of SOA services

This paper focuses on steps 2 and 3, and touches on step 4 of this process. Step 2 involves the creation of a model of the software system. Step 3 involves recognizing the elements of the legacy system which expose the functionality desired by each service in the SOA system, and performing a dependency analysis on those elements to identify the portion of the legacy system that implements that functionality (the ‘support set’ of that functionality). Step 4 involves the identification of portions of the legacy system which help implement multiple identified sets of functionality.

In this research we assume that step 1, the identification of the set of desired services, is completed by the analysts and engineers of the legacy system, using other methods. We do not provide an algorithm for intersecting support sets or for partitioning the system model, and we do not address the details of reengineering the legacy software system. We produce graphs which represent the aforementioned support sets in a way which enables the use of graph algorithms to identify support set intersections.

The process necessitates a model of the legacy software system, and a way to perform dependency analysis at a potentially abstract level upon that model. We determine the appropriate kind of model to represent system elements and dependencies with, and developed an analytical technique, graph slicing, to derive support sets from information represented in this model.

We develop a software framework, the Relationship Modelling Framework (RMF), which is capable of producing and maintaining the kind of model on which our analytical technique operates.

4 Relationship Modelling

This section describes the models we use to perform relationship modelling, the framework we developed to generate these models, and the analytical technique used to compute support sets based on these models.

4.1 Semantic Graphs

We represent semantic models in the RDF language. However, the concepts produced in this research are applicable to any directed-graph-based representation of a software system. For this reason, we discuss semantic graphs, rather than RDF graphs in specific. A semantic graph is any directed graph which represents elements of a software system, and relationships between those elements.

The use of a graph to represent information about a software system assumes that all of the queries needed to perform a system decomposition can be posed against a graph. We use this assumption as the basis of our solution to the decomposition problem.

4.2 Graph Slicing

A graph slice of a directed graph $G$, with respect to a set $S \subseteq V(G)$, is the graph induced by the vertices in $S$ and all of their descendant edges and vertices.

Graph slicing can model other forms of slicing, including program slicing. For example, given a graph $G$ whose vertices represent statements and variables
in a procedural program, and whose edges represent control- and data-flow dependencies, a graph slice of \( G \) with respect to a vertex set \( S \) is equivalent to a program slice isolating the statements and variables represented by the vertices in \( S \) (based on the example program slice given in 2.1.1).

In general, if the vertices of a graph \( G \) represent functional elements of a software system, and the edges represent functional dependencies between those elements, then a slice of \( G \) with respect to a set of vertices \( V \) represents a component of the software system which implements all of the functionality exposed by the vertices in \( V \). The identified functional elements can then be extracted from the software system and used to implement that functionality independently from the rest of the system.

4.3 The Relationship Modelling Framework (RMF)

RMF is a semantic graph generation, maintenance and querying framework. It specifies the format of the semantic graph, a layered organization for abstracting the steps of the model generation and query process, and a modular organization for grouping and distributing the modelling components. It also specifies a management structure for managing modules and their interdependencies, for persisting and maintaining the semantic graph, and for persisting and managing system properties.

RMF generates and maintains semantic graphs based on the electronic documents which describe a software system, such as source code, object code, and configuration files.

RMF uses the RDF specification for storing and representing a semantic model. All information generated by the implementation is collected together and represented centrally in the same RDF graph.

RMF consists of three layers, the extraction, generation and query layers. The extraction layer parses and processes documents describing the legacy software system to extract information about the system. The generation layer uses this information to construct or update the semantic graph. The query layer performs post-processing on the semantic graph and produces the results for further analysis.

RMF is a software framework. It does not generate the semantic model itself; it relies on modules to extract information and generate the semantic model. Modules are interdependent sets of modelling components, each of which handles a particular domain of modelling capability. Examples of modules include a particular kind of document (such as an XML parsing/querying module), or a particular class of information (such as class inheritance information).

The architecture of RMF is shown at a high level of abstraction in figure 3. The diagram shows four management objects: the graph manager, property manager, command manager and strategy manager. The graph manager manages the semantic graph produced by the framework. The property manager manages persistent configuration information used by the framework and its component modules. The command and strategy managers control access to components in the generation and extraction layers, respectively. One more management object, the module manager, is not shown in this diagram. The module manager loads, unloads and manages dependencies for the modules which populate the extraction and generation layers.

5 Prototype and Validation

To validate our hypothesis, we developed a prototype of RMF to perform graph slicing on a semantic graph generated from a production-quality large software system. This prototype differs in one chief way from the system described in the preceding section: it is not capable of maintaining the semantic graph. It must generate a separate graph for each query made.

The analysed software system was a large commercial suite based on Java’s J2EE system [17], designed and developed by IBM. The company has requested that the software suite remain anonymous; we will identify it as the ‘large J2EE software suite’.

The software suite consists of approximately 5
million lines of code, written exclusively in the Java programming language. It makes extensive use of non-Java-code resources such as XML configuration files, code generation tools, and SQL database scripts in the form of stored database procedures and text files, as well as embedded in Java strings.

The architecture of the J2EE suite at a high level of abstraction is shown in figure 4. Table 1 gives several statistics about the system.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code</td>
<td>(\approx 5,000,000)</td>
</tr>
<tr>
<td>Java classes</td>
<td>(\approx 29,500)</td>
</tr>
<tr>
<td>Total number of Command Interfaces</td>
<td>(\approx 4,500)</td>
</tr>
<tr>
<td>Total number of Command Impls</td>
<td>(\approx 4,500)</td>
</tr>
<tr>
<td>Total number of Access Beans</td>
<td>(\approx 600)</td>
</tr>
<tr>
<td>Total number of EJBs</td>
<td>(\approx 600)</td>
</tr>
<tr>
<td>Total number of EJB User Finders</td>
<td>(\approx 300)</td>
</tr>
<tr>
<td>Total number of Database Tables</td>
<td>(\approx 600)</td>
</tr>
</tbody>
</table>

Table 1: Statistics regarding the J2EE suite

6 Results

Three graph slices were generated based on the J2EE software suite. For each slice, several nodes were chosen at random to be measured for accuracy; additionally, five EJBs from the J2EE suite were chosen based on a complexity estimate made by IBM engineers, and measured for each graph in which they appeared. Five EJBs were chosen because of the amount of work required to analyse each, and because the J2EE suite engineers considered these five to be reasonably representative of the complete set of EJBs. Tables 2 and 3 summarize the results.

Table 2 displays the time taken to generate the semantic graph and the accuracy of the tested vertices for each of the three slices produced. The time taken is separated into two components in order to highlight that the majority of the time needed was spent recognizing one type of dependency—identifying implementations for a given ‘Command’ interface. Table 3 displays the overall accuracy of the analysis, separated into false positives (graph elements inaccurately retained) and false negatives (graph elements inaccurately omitted).

The graph generation process required almost 2 hours in the slowest trial, and almost 7 minutes in the fastest trial; in the two longer trials, approximately 97% of the time was spent finding implementations of the entities and relationships described in figure 4.

The RMF prototype and the modules we developed were given to engineers at IBM for the validation process. The engineers selected three ‘Command Interface’ elements from among those present in the J2EE suite, based on a combined complexity and diversity estimate. For each element they used the prototype and modules to produce a graph slice with respect to the ‘Command Interface’ element itself. Because of the prototype’s inability to persist the semantic model between query executions, we designed the query to generate only the portion of the semantic model which constituted the graph slice, rather than generate the semantic model for the entire J2EE suite and perform the graphic slice on it.

The performance of the graph generation process was recorded, and the resulting dependency graphs sampled both randomly and selectively. Each sample was compared to the results of a manual inspection of the relevant system documents to produce a measurement of the accuracy of the slice with respect to the implemented J2EE system.
Table 3: Overall accuracy

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total false positives</td>
<td>0</td>
</tr>
<tr>
<td>Total false negatives</td>
<td>11</td>
</tr>
</tbody>
</table>

for ‘Command’ interfaces. The random sample offered perfect accuracy in all 3 trials. Over the five selected EJBs, 29 database table references were identified during the manual inspection. In the first trial, all five EJBs were used, but only 25 of the 29 database table references were identified. In the second trial, only four of the five EJBs were used, and only 23 of the 26 relevant database tables were linked. In the third trial, a different set of four EJBs were used, and only 19 of the 23 relevant tables were linked.

The performance of the graph generation process, requiring almost 2 hours in the worst case and almost 7 minutes in the best case, implies that it is not feasible to generate the graph for ‘online’ processing. This limitation is almost entirely caused by the cost of identifying implementations for a given interface, an operation which requires scanning the roughly 30,000 classes comprising the J2EE suite. It can therefore be mitigated by implementing a more efficient algorithm for performing this analysis; however it should be noted that, given an RMF implementation that persists the semantic graph across invocations and incrementally updates it when the underlying system is changed, the cost to generate the semantic graph as a whole would only be incurred once over the lifetime of the legacy software system.

The false negatives shown in table 3 are particularly interesting. When testing the algorithm for identifying database tables on which an EJB depends, we encountered no errors of this type. After encountering these errors, we took the same EJBs that caused a problem for the graph slicing analysis, and executed the algorithm directly on those EJBs. When the database table dependency algorithm was executed on the EJBs directly, it gave 100% accuracy rate; the same EJBs still showed false negatives when analysed in the context of the graph slicing query, even though the same code was used in both cases. This suggests that the inaccuracies detected are a result of a coding error, rather than of a limitation of the algorithm itself.

We identified one issue which may affect how representative the results are of software systems in general. IBM emphasizes architectural adherence. The J2EE system engineers may adhere more closely to architectural constraints and best practises than engineers of other systems. If this is the case, then the development of modelling components for an RMF implementation may be more difficult for other software systems than for the J2EE suite.

7 Conclusion

The purpose of this research was to study the application of a semantic model to modelling a software system comprised of many different kinds of document, and the automatic processing of such a model to provide insight into the structure of the modelled system for the purpose of software decomposition. Our research represents a first step towards a larger goal: to determine the utility of such a system for supporting software reengineering, and software engineering in general.

Our research demonstrates that the RMF architecture can generate a semantic model of a software system comprised of many different kinds of document. It also provides evidence that, using the graph slicing algorithm, we can automatically produce dependency graphs which approximate a functional subset of the underlying software system.

8 Future Work

The Relationship Modelling Framework prototype displays two notable limitations which are clearly specific to the prototype. The first limitation is implied by the amount of time needed to generate the semantic graph, in preparation for the graph slicing operation. This is necessary because the prototype does not maintain the semantic graph; for each query execution the graph must be regenerated. If the semantic graph were maintained, and updated when the underlying system was modified, virtually all of the cost associated with performing a graph slice would be eliminated (except for the first operation, at which point the graph would be generated and retained for future use).

The second limitation is in the accuracy of the EJB-to-database table dependency identification. Although the demonstrated accuracy is reasonably good (approximately 86%), it could be improved upon by using a better dependency identification algorithm. We have identified several ways in which the EJB-to-database dependency identification algorithm could be improved upon (most notably by performing a more complete control- and data-flow dependency anal-
ysis on the Java procedures which produce and/or execute the SQL statements).

The RMF architecture needs to be developed further. In particular, a mechanism for automatically generating semantic graph elements, rather than requiring a query to explicitly update or generate them, would be valuable.

Further study of the goal-oriented decomposition process is needed, both to establish its usefulness and to place it among other reengineering procedures that could be undertaken to achieve similar results. In this context, methods for partitioning service support sets into services which embody the maxims of loose coupling, strong cohesion and code re-use need to be studied.

After establishing the utility of a semantic model for graph slicing, a study of other areas which such a model could support would be valuable, such as checking or maintaining artifact consistency, visualization, software design, and software testing. IBM engineers have identified a number of additional use cases for the RMF architecture based on the results of this research, such as documentation generation, customization support, defect location and developer education.

References


Table 2: Result summary

<table>
<thead>
<tr>
<th></th>
<th>6,735.3s (≈112m)</th>
<th>392.3s (≈6.5m)</th>
<th>2,863.6s (≈48m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time to execute</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“Command” implementation identification</td>
<td>6,580.7s (97.5%)</td>
<td>334.1s (85.1%)</td>
<td>2,778.0s (97.0%)</td>
</tr>
<tr>
<td>Other graph generation tasks</td>
<td>154.7s (2.5%)</td>
<td>58.2 (14.9%)</td>
<td>85.5 (3.0%)</td>
</tr>
<tr>
<td>Random sample accuracy</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Selective EJB accuracy</td>
<td>25/29 (86.2%)</td>
<td>23/26 (88.5%)</td>
<td>19/23 (82.6%)</td>
</tr>
</tbody>
</table>


