A Service Sharing Approach to Integrating Program Comprehension Tools

Dean Jin James Cordy

School of Computing
Queen's University, Kingston, Canada
{jin,cordy}@cs.queensu.ca

Keywords: program comprehension, software understanding, tool integration, maintenance, concept support, service sharing, domain ontology

1. Introduction

Software maintenance is the most time consuming and costly phase of the software development lifecycle. For every dollar spent on creating a new software system, nine dollars is spent on maintaining it throughout its useful life. By the late 1980s maintenance spending accounted for an estimated US$30 billion worldwide. Any activity that even minimally reduces maintenance efforts would yield significant cost savings within the software industry [3].

Tool support for maintainers has focused largely on providing assistance in activities related to program comprehension. The goal of these tools is to provide a rapid means for maintainers to understand large scale software systems. Most program comprehension tools have a specific strength or specialized application area [10] but are weak in other areas. No single tool exists that provides all the functionality and flexibility that most software maintainers need. For this reason, research attention has been focused on getting program comprehension tools to integrate with each other.

In this paper we present a novel approach to facilitating integration among tools used by maintainers to assist in program comprehension. We start by showing that program comprehension tools have many similar characteristics. Taking full advantage of this fact, we outline how specially designed adapters and a domain ontology can be used together to allow these tools to integrate transparently with each other.

2. Program Comprehension Tools

Program comprehension tools provide a means for maintainers to understand a software system from a functional and behavioural perspective [11]. Functional comprehension provides insight into what the system does. Behavioural comprehension highlights how a system works. These tools support a variety of activities carried out by maintainers including:

- *Disintegration.* Breaking larger systems into subsystem components.
- *Pattern Matching.* Identification of instances within the source code where an identical coding pattern occurs.
- *Program Slicing.* Isolation of all code that relates to or in some way impacts the execution of a specific point in the source code.
- *Dependency Analysis.* Evaluation of the reliance of system components on other internal or external components.
- *Metrics Evaluation.* Measurement of the code according to accepted standards for various characteristics such as size, complexity, quality, maintainability, etc.
- *Exploration.* Support for navigation throughout the source code.
- *System Visualization.* Generation of views for examining the system visually.

In many instances, a tool provides assistance but still requires significant manual intervention on the part of the user.

A tool *service* is the functionality provided by a tool that, when given a set of one or more inputs, generates a corresponding output that is relevant for maintainers. In the case of program comprehension tools the inputs are typically source code (or facts about the source code) and the output is typically a report or visualization.

3. Hypothesis

Previous approaches to program comprehension tool integration have been *data centric*, concentrating on the exchange of data through rigid standardized formats or spe-

cialized hard-coded tool interfaces. This kind of integration is *prescriptive*. In essence, it forces tool developers to conform to an idiomatic standard or provide a particular functionality to other tools in order to participate in the integration process.

The hypothesis of our work is that integration among a set of program comprehension tools can be accomplished more effectively using external, tool-independent adapters that make use of a common vocabulary of concepts shared among the tools. This kind of integration focuses on sharing the services offered by each tool rather than simply exchanging data among them.

The novel integration methodology we outline in this paper is distinctly *non-prescriptive*. Instead of forcing tool conformance, we take advantage of commonalities that exist among tools. We do this by focusing on the operational and representational concepts that each tool exhibits. We demonstrate that working from this conceptual perspective allows us to fully exploit the similarities that exist between tools to facilitate integration.

We start by observing how program comprehension tools are similar in terms of their architectural makeup and operational characteristics.

4. Tool Architecture

Although many program comprehension tools exist, most feature the same underlying architecture and functionality [3]. In general, program comprehension tools consist of the following three interrelated components [1, 7, 8, 12]:

1. **Fact Extractor.** On the front end, program comprehension tools typically input source code and extract facts from it.

2. **Repository.** The quantity of facts extracted from source code can be substantial. For this reason, it is typically organized and physically stored in a structured format such as a database rather than preserved in memory.

3. **Analyzer/Visualizer.** The facts in the repository are processed and analyzed with the results presented visually or through reports.

5. Operational Characteristics

Elmasri and Navathe [6] define a database as a collection of related, recordable facts with implicit meaning. This collection, along with software that manages and manipulates the collection make up a *database system*. Considering the architecture of program comprehension tools outlined in Section 4, it is readily apparent that they are in fact database systems specially tailored to store, manipulate and analyze software facts.

From this database systems perspective we can abstract the operational characteristics of program comprehension tools into three distinct layers:

1. **Transactions.** The queries and updates that extract, process and analyze the software facts stored in the database.

2. **Schema.** A definition for the entity types, relations and constraints that make up the information model used by the tool to represent software. Similar to database systems, most program comprehension tools use *Entity-Relationship* (ER) [2] models to define their schemata.

3. **Instance.** Software facts stored in the database in a form defined by the schema on which the tool transactions operate. For the purpose of our discussion, we refer to a program comprehension tool database populated with software facts as a *factbase*. The instance for a given program comprehension tool is simply the factbase that the tool populates, manipulates and maintains.

6. Service Sharing

The similarities among program comprehension tools in terms of architecture and operational characteristics provide a significant advantage from an integration perspective. Typical integration efforts involve the resolution of broadly different operational paradigms. Within the restricted domain of program comprehension tools, the very difficult problem of reconciling operational differences is largely nonexistent. This leaves service sharing as the remaining integration challenge to be dealt with.

Three issues relate directly to sharing services among program comprehension tools: *concept support*, *transaction application* and *representational diversity*. We now discuss each of these issues in more detail.

6.1. Concept Support

The notion of *concept* as it relates to factbases is an important part of our discussion on service sharing. Maintainers use program comprehension tools to extract knowledge about software from its representation stored in a factbase. The knowledge that a given factbase can provide depends on the concepts that the representation supports. We call this *concept support*.

In another paper [9] we provide a detailed discussion of concept support among program comprehension tools. We classify concept support in a given factbase as follows:

- **Native.** The factbase explicitly supports the representation of the concept. Other than possible differences in the names used, a complete representation for the concept exists in the factbase.

- **Derived.** The factbase supports the representation of the concept, but it must be derived or inferred from the facts represented. A query can be constructed that extracts an equivalent representation from the factbase.

- **Undefined.** The factbase is fundamentally incapable of representing the concept. This means that no information content for the concept is available in the factbase. Provided the absence of certain facts related to the concept can be tolerated, a partial representation may be available.

A program comprehension tool factbase can support any number of concepts. Nevertheless, it is important to keep in mind that not all concepts are supported in all factbases. For integration based on service sharing to work, all factbases participating in the integration must include some kind of support for the concepts that a particular service operates on.

6.2. Transaction Application

A program comprehension tool provides a service to maintainers by executing one or more transactions on software facts stored in their factbase. The actual implementation of each of these transactions depends completely on the factbase structure. Since the structure of the factbase is itself dictated by the schema, the implementation of each transaction completely depends on the schema as well.

In order to share tool services, we must find a way of applying transactions (whose implementations are specific to a particular tool) to factbases from other tools. Two methods can be considered:

1. **Transaction Translation.** This is the 'bring the transaction to the data' method. Each transaction implementation is translated into a new implementation that works with the factbase for another tool.

2. **Fact Provision.** This is the 'bring the data to the transaction' method. The required facts are extracted from another factbase, appropriately formatted and provided as input to an existing transaction implementation.

Deciding which approach to use for transaction application is a matter of evaluating the tradeoffs between transaction implementation complexity and the work involved in extracting, formatting and working with existing transaction implementations.

6.3. Representational Diversity

A real challenge in sharing tool services relates to reconciling the diversities that exist in the representation of software knowledge that each tool maintains. These differences relate to issues of *syntax* and *semantics*.

Syntax

Structural differences in the way software facts are manipulated and stored account for the syntactic divergencies that exist between tools. In relation to integration, we are concerned with transforming software facts represented in one form to a corresponding form acceptable by another tool. This transformation must be *isomorphic*. Nothing can be added to or taken away from the facts. Only the structural characteristics of the facts can be changed. For the most part, syntactic differences are easily reconciled through representational mapping and translation operations.

Semantics

The most difficult aspect of integration that must be addressed is how to deal with *semantics* or differences in meaning among the software facts maintained by each of the program comprehension tools.

No single information model captures all the views of software supported by all program comprehension tools currently available [5]. This is because a myriad of semantic differences exist between models for programming languages [4]. For example, in object-oriented languages such as Java all entities are organized within a hierarchy of classes. Instantiation outside the class hierarchy is not possible. In contrast, modular languages such as COBOL allow the creation of global external variables and records; entities which are nonexistent in object-oriented programming languages.

In addition, many semantic differences stem from the use of program comprehension tools in various application domains. For example, software that supports financial systems, user interface systems and scientific computing systems all have unique characteristics whose meaning is represented differently, depending on the program comprehension tool being used.

7. Design Decisions

The goal of service sharing is to facilitate integration among program comprehension tools in a transparent manner that capitalizes on the similarities between them. To accomplish this, we make use of a *domain ontology* to organize the various services, concepts and syntactic characteristics that each tool offers to the integration.

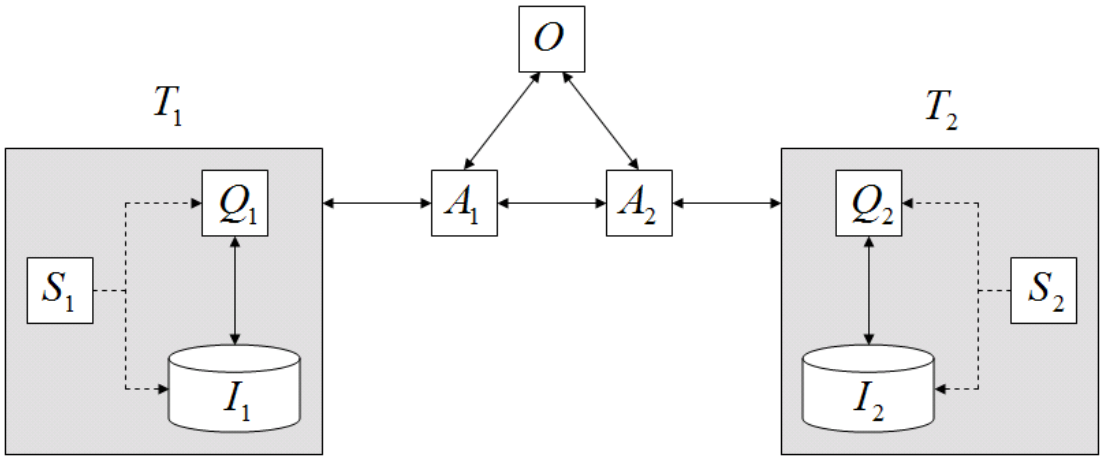

Figure 1. An architecture for sharing services among two program comprehension tools

To keep our solution as simple as possible we keep the integration components separate from the tool implementation and instead adopt a 'plug-in' approach to participation. Our intention is to allow developers complete autonomy in their tool creation efforts. We believe the best way to promote integration is to make it as easy as possible to plug into an existing integration environment rather than try to build one from scratch.

One way to maximize the effectiveness of the integration and minimize the effort involved in constructing the domain ontology is to apply our methodology only to program comprehension tools. This allows us to confidently exploit the the similarities among program comprehension tools in terms of architecture and operational characteristics that we discussed in Sections 4 and 5. It also allows us to make a fundamental assumption: Although there may be vast differences in the way each factbase is structured, there is likely a significant amount of equivalence among the concepts supported by tools participating in the integration. There is likely a limit to the variation in concept support that tools in the same application domain exhibit. This means that once the domain ontology is initially built, it should be easier to add participants to the integration at a later date.

The design of our integration methodology was driven by the need to address each of the issues related to service sharing. For instance, we originally chose to make use of a domain ontology to keep track of concept support for each tool participating in the integration. It soon became apparent that the ontology could be used to manage all the knowledge about services, concepts and syntactic characteristics that are required to facilitate integration. This ontology provides a common vocabulary for tool adapters that operate very much like software agents in the way they to carry out integration tasks.

An assessment of the complexities involved in translating transactions for the tools we wanted to integrate led to our decision to use fact provision as opposed to transaction translation.

Our solution also addresses the issue of syntax in relation to representational diversity. Integration adapters use the knowledge of syntax maintained in the domain ontology to apply syntactic converters in situations where there is a structural difference in facts represented by tools in the integration.

Our integration methodology cannot directly address the issue of semantics as it relates to representational diversity. This requires the skill and knowledge of a human being. Nevertheless, our use of a domain ontology and the separation of integration components from tool implementation details emphasizes efficiency and simplicity. This should make it easier to construct the domain ontology and maintain the integration environment.

8. An Ontological Approach

Figure 1 provides an architectural view of our shared service integration methodology. Here we see two participant tools T_1 and T_2 involved in an integration. Each tool has a set of transactions (Q_1 and Q_2), a schema (S_1 and S_2) and a correspondingly structured instance (I_1 and I_2). Our implementation involves the creation of two types of components:

1. **Domain Ontology** (O). All the knowledge required to support service sharing among each of the tools participating in the integration is stored here. It is essentially a tabularized, cross referenced compilation of

shared services, concepts and syntactic characteristics supported by each tool. Only one domain ontology is required for the implementation.

2. **Conceptual Service Adapters** (A_1, A_2). These operate as an integration facilitators for tools participating in the integration. One adapter is affiliated with each integration participant. The adapters make extensive use of the domain ontology to get the information they need to facilitate interoperability among integration participants. Each adapter performs the following four main functions:

   (a) *Shared Service and Concept Support Identification.* Making use of the knowledge of services stored in the domain ontology, each adapter identifies requests for shared services and determines the concept support each service requires.

   (b) *Information Extraction.* Using knowledge of concepts stored in the ontology, each adapter extracts facts related to the concepts required for the shared service from the factbase.

   (c) *Syntax Conversion.* Making use of the syntactic knowledge stored in the domain ontology, each adapter looks after structuring facts so that they conform to the schema for the tool where the service being shared is implemented. They also look after converting the results returned back from the shared service.

   (d) *Shared Service Execution.* Each adapter manages requests from other conceptual service adapters for the execution of shared services on the tool they are associated with.

The domain ontology is instrumental in providing the conceptual service adapters with the knowledge they need to facilitate interoperability among tools participating in the integration. For this reason it is important that the greatest care be taken to ensure that the ontology constructed is as comprehensive and complete as possible. In another paper [9] we describe the construction of the domain ontology in greater detail. The long term benefits of using the ontology far outweighs the short term pain involved in creating it.

A key attribute of each conceptual service adapter is *transparency*. In essence, each adapter tricks their associated tool into thinking it is performing a service on its own factbase. In reality, the facts provided are from a factbase from one of the other participants in the integration. Neither tool is aware that the adapters are acting as liaisons between them. Successfully implemented, the conceptual service adapters provide shared services among all participants in the integration in a seamless, completely transparent manner.

9. How It Works

Consider two program comprehension tools T_1 and T_2 as shown in Figure 1. T_2 offers a shared service V which we would like to apply to the factbase of T_1. The domain ontology O has been constructed based on the services, concepts and syntactic characteristics supported by T_1 and T_2. The conceptual service adapters A_1 and A_2 are now ready to facilitate the interoperability we need to achieve our goal.

The request for service V invoked from T_1 is received by A_1. The adapter uses the domain ontology to identify V as a shared service offered by T_2. It also learns that V requires a factbase that supports concepts c_3, c_{21} and c_{44}. A_1 accesses O and verifies that T_1 supports these required concepts. It then extracts the facts from I_1 that correspond to concepts c_3, c_{21} and c_{44} and syntactically converts them into a form compliant with S_2 using the syntax information provided by O.

A_1 then sends a request to A_2 asking it to execute shared service V on the S_2 compliant facts. A_2 returns the results of the execution of shared service V back to A_1. The results are syntactically converted from the S_2 compliant form back to the S_1 compliant form and returned to T_1.

The integration facilitated by the conceptual service adapters is completely transparent. We have applied shared service V to facts from the I_1 factbase just as though they were facts from I_2.

10. Conclusion

Work on this project is a continuing effort leading toward the implementation of an integration of four program comprehension tools. We are interested in demonstrating small-scale integration to help with the initial construction of a domain ontology that we hope to make available in a later publication. The experiences gained from a prototype integration among two program comprehension tools has contributed significantly to our understanding of the challenges and benefits offered by integrating tools that support program comprehension.

Diversity among program comprehension tools is good thing because it leads to the development of new and innovative solutions to problems that are directly relevant to software maintainers. As we mentioned in the introduction, the main barrier to more widespread application of tools that support maintenance tasks is the lack of integration that exists between them.

In this paper we have presented a new integration methodology for program comprehension tools. We started by showing that program comprehension tools have many similarities in their architecture and operational characteristics. Taking full advantage of this, we outlined how specially designed adapters and a domain ontology can be used

together to allow these tools to integrate transparently with each other.

Acknowledgements

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) as part of the Consortium for Software Engineering Research (CSER).

References

[1] I. T. Bowman, M. W. Godfrey, and R. C. Holt. "Connecting Architecture Reconstruction Frameworks". In *Proceedings of the 1st International Symposium on Constructing Software Engineering Tools (CoSET'99)*, pages 43–54, Los Angeles, CA, May 1999.

[2] P. Chen. "The Entity Relationship Model – Toward a Unified View of Data". *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[3] E. J. Chikofsky and J. H. Cross II. "Reverse Engineering and Design Recovery: A Taxonomy". *IEEE Software*, 7(1):13–17, January/February 1990.

[4] S. Demeyer, S. Ducasse, and S. Tichelaar. "Why FAMIX and not UML?". In *Proceedings of UML'99*, volume 1723 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[5] J. Ebert, B. Kullbach, and A. Winter. "GraX: Graph Exchange Format". In *Proceedings of the Workshop on Standard Exchange Formats (WoSEF) at ICSE'00*, Limerick, Ireland, 2000.

[6] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 3rd edition, 2000.

[7] M. W. Godfrey. "Practical Data Exchange for Reverse Engineering Frameworks: Some Requirements, Some Experience, Some Headaches". *Software Engineering Notes*, 26(1):50–52, January 2001.

[8] R. C. Holt, A. Winter, and A. Schürr. "GXL: Toward a Standard Exchange Format". In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00) Panel on Reengineering Exchange Formats*. IEEE Computer Society Press, November 2000.

[9] D. Jin, J. R. Cordy, and T. R. Dean. "Transparent Reverse Engineering Tool Integration Using a Conceptual Transaction Adapter". In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR 2003)*, pages 399–408, Benevento, Italy, March 2003.

[10] T. C. Lethbridge. Requirements and Proposal for a Software Information Exchange Format (SIEF) Standard, November 1998. URL: `http://www.site.uottawa.ca/~tcl/papers/sief/standardProposalv1.html`.

[11] S. R. Tilley, S. Paul, and D. B. Smith. "Towards a Framework for Program Understanding". In *Proceedings of the 4th International Workshop on Program Comprehension (IWPC'96)*, pages 19–28, Berlin, Germany, March 1996.

[12] S. Woods, L. O'Brien, T. Lin, K. Gallagher, and A. Quilici. "An Architecture For Interoperable Program Understanding Tools". In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC'98)*, pages 54–63, 1998.