Proc. CSMR 2003, IEEE 7th European Conf. on Software Maintenance and Reengineering, Benevento, Italy, March 2003, pp. 399-408.

Transparent Reverse Engineering Tool Integration Using a Conceptual Transaction Adapter

Dean Jin James R. Cordy Thomas R. Dean

Queen's University, Kingston, Canada

{jin,cordy}@cs.queensu.ca, thomas.dean@ece.queensu.ca

Abstract

In this paper we present a proposal for a novel approach to facilitating transparent interoperability among reverse engineering tools. We characterize the architectural and operational characteristics of reverse engineering tools and demonstrate that many similarities exist among them. Taking full advantage of these similarities, we outline an approach for creating a domain ontology of operational and representational concepts for a given set of tools. A special adapter is proposed that makes use of this ontology to facilitate transparent interoperability among them.

1. Introduction

The past decade has seen an increased awareness of the challenges presented by the maintenance phase in the software development lifecycle. Software maintenance involves many tasks of which *reverse engineering* [3] is one of the most important. It involves analyzing a software system to determine how it is constructed, resulting in the creation of representations that aid in system comprehension.

Various tools designed to assist maintainers in carrying out reverse engineering have been created. Most of these tools have a specific strength or specialized application area [23] 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 reverse engineering tools to interoperate with each other.

Successful tool interoperation involves both technical issues (such as inter-tool communication, sharing protocols, distributed update management, etc.) and information issues (such as representational diversity, incompleteness, equivalency, etc). In this paper we restrict our discussion to the latter, with special emphasis on sharing tool services as opposed to simply sharing data. Technical issues will be the focus of our implementation efforts and future papers.

We present a proposal for a novel approach to facilitating transparent interoperability among reverse engineering tools. We start by showing that reverse engineering tools have many similar characteristics. Taking full advantage of this fact, we outline how a specially qualified adapter and a domain ontology can be used together to allow reverse engineering tools to interoperate seamlessly.

2. Why Interoperability Matters

While progress has been made towards increasing the performance and usefulness of reverse engineering tools, most continue to exist in isolation, lacking any effective means for sharing information among each other [7, 30]. This lack of integration has been a serious obstacle to the adoption and use of automation in software maintenance tasks [26, 20].

Analysis from different tools can help speed up the reverse engineering process [24, 31]. For example, Holt, Winter, Schürr and Sim [21] list eighteen examples of reverse engineering, software modelling, analytical and graphing tools that provide many different types of analyses that reverse engineering practitioners can make use of. Maintainers can leverage their results by combining the output of different analyses from different tools [1, 15, 22, 27, 29]. Tool interoperability would allow software maintainers the opportunity to use a suite of tools; each specialized for a particular maintenance task.

Without an integrative component, maintainers are forced to work independently with each tool starting from scratch [1, 22]. Manually integrating results from different tools is tedious and time consuming [6, 30, 31].

To effectively assist in maintenance tasks, reverse engineering tools must be able to work together to provide a deep and consistent understanding of the systems involved.

3. Barriers To Integration

The primary barrier to reverse engineering tool interoperability is differences in the representation of software

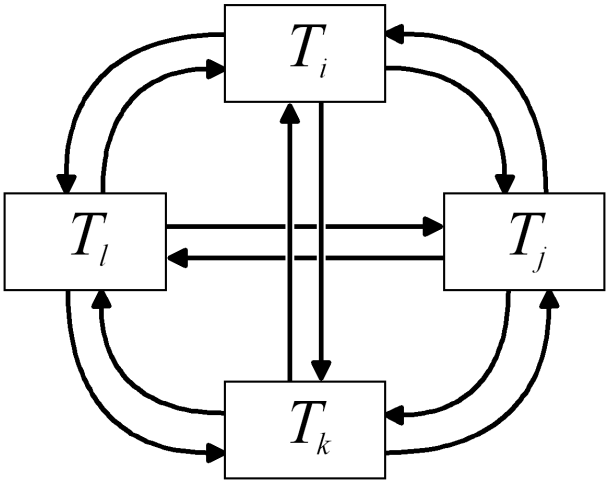

Figure 1. Integration Utopia (4 Tools)

knowledge that each tool maintains. These differences are both *syntactic* and *semantic*. Structural differences in the way information is manipulated and stored account for the syntactic divergencies that exist between tools. For the most part, syntactic differences can easily be reconciled through representational mapping and data translation.

Semantic differences are more difficult to resolve. No single information model captures all the views of software supported by all reverse engineering tools currently available [8]. 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 reverse engineering 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 reverse engineering tool being used.

It is clear that a minimal requirement for achieving tool interoperability involves the mediation of the syntactic and semantic particularities inherent among reverse engineering tools. This points to a solution that operates above the level on which data is represented.

4. Integration Utopia

Today, reverse engineering tool integration is most often implemented on a tool-by-tool basis. Special converters are used to transform information from one tool into a form that is compatible with another tool. This kind of integration is less than ideal for a number of reasons:

- Creating special converters is time consuming and often targets only a specific application.
- The incentive to integrate with other tools is low. It is usually not worth the effort to create a converter for such a special purpose.
- It leads to the creation of proprietary integration solutions. These are hard to maintain and make it difficult for other tool developers to participate in the integration.

What we refer to as *integration utopia* is full interoperability among n reverse engineering tools. Each tool can apply its analyses on the data of all the other tools and vice versa. Integration utopia among four reverse engineering tools is shown in Figure 1. Using current integration technology, full interoperability among n reverse engineering tools would require $n(n-1)$ converters. Clearly a different approach to facilitating integration among reverse engineering tools is required before integration utopia is achievable.

5. Reverse Engineering Tool Architecture

Although many reverse engineering tools exist, most feature the same underlying architecture [3]. In general, reverse engineering tools consist of the following three components [1, 5, 15, 20, 33]:

1. **Information Extractor.** The front ends of reverse engineering tools typically input source code and extract information from it. A lexer reads the code and breaks it into lexical tokens. These tokens represent the keywords and basic building blocks for the program based on the specific programming language that is being used. Next, a parser groups the lexical tokens into programming constructs like statements, expressions, declarations, etc. These facts are almost always organized into a graph structure. Some tools use an Abstract Syntax Tree (AST) or an Abstract Semantic Graph (ASG) to represent software facts.

2. **Repository.** The quantity of information extracted from source code can be substantial. For this reason, information extracted from the source code is typically organized and stored in a repository rather than preserved in memory.

3. **Analyzer/Visualizer.** The information in the repository is processed and analyzed with the results presented visually, through reports or by source code markup.

6. Operational Characteristics

Elmasri and Navathe [9] 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 basic architecture of reverse engineering tools outlined above, it is readily apparent that they are in fact database systems specially tailored to store, manipulate and analyze information about software. On the front-end, parsers provide structured facts that 'populate' the database. At the back-end, visualizers and analyzers make use of facts in the database to yield information that is useful to maintainers. Between these two extremes lies a management system which provides the means for defining, constructing and manipulating the database.

Continuing from this database systems perspective we can abstract the operational characteristics of reverse engineering tools into three distinct layers as shown in Figure 2:

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 reverse engineering 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 reverse engineering tool database populated with software facts as a *factbase*. The instance for a given reverse engineering tool is simply the factbase that the tool maintains.

Approaches to facilitating interoperability among reverse engineering tools have so far concentrated on negotiating the transfer of information at the schema and instance levels. These efforts have essentially been an ad hoc exercise in *data integration*; an attempt to get information stored in one tool into a form that another tool can use [32]. Although this approach is effective for bridging the exchange gap from a syntactic perspective, it falls short in dealing with differences among the various data structures, instance semantics and information models that each tool employs.

7. Conceptual Transactions

Many reverse engineering tools use the same or similar sets of transactions to analyze factbases. For example, a dependency analysis might see two tools querying their

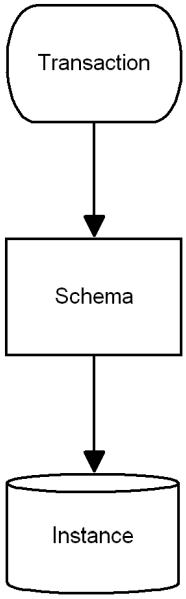

Figure 2. Operational Characteristics of Reverse Engineering Tools

factbases for 'reference' relations. In one tool this might be as simple as querying the factbase for all entities that have a 'refers-to' relation to another entity. In another tool this might involve an exhaustive search through all source code references for variables that exist in other modules.

From an implementation perspective, each tool is radically different in the way it represents reference relationships. The first tool explicitly records reference relations so they are easily obtained by a simple query. In the second tool the same information is stored implicitly, so extraction involves pattern matching to filter out the desired results. Nevertheless, both tools use the same *conceptual transaction* to get their results; namely the identification of dependency among entities in a representation for software. Direct correspondence or similarity among reverse engineering tool transactions is the key to our proposal for a new conceptual transaction-based integration paradigm.

The notion of *concept* as it relates to factbases is an important part of our discussion on conceptual transactions. Maintainers use reverse engineering tools to extract *knowledge* about software from its representation stored in a factbase. The knowledge that a given factbase provides depends on the concepts that the representation supports. The example we provided above focused on the concept of dependency in two reverse engineering tool factbases. A factbase in a reverse engineering tool can support any number of concepts. Nevertheless, it is important to keep in mind that not all concepts are supported in all factbases. A prerequisite for integration using conceptual transactions is that

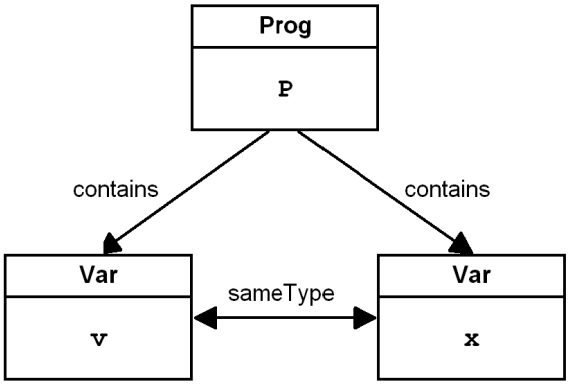

Figure 3. Native Support for the ‘same type’ Concept

all participant factbases support the concept that the transaction refers to.

Concept support in a given factbase can be classified 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.

In Figures 3 to 5, three very small hypothetical representations of software facts are used to demonstrate the difference between each concept support category. Our goal in these examples is to identify all instances of the ‘same type’ concept in the representation. Native support for the concept is shown in Figure 3. Here an explicit sameType relation is defined between variable **v** and variable **x**. In Figure 4 the ‘same type’ concept is not explicitly represented, but it is obtainable from the facts available. We see that variables **v** and **x** are both **array** types. It is not clear that these variables have the same type until each of their elementType relations are found to lead to the same **int** type node. A query that reconciles the type of array structures could be used to support the ‘same type’ concept for this representation. In Figure 5 no information about the type

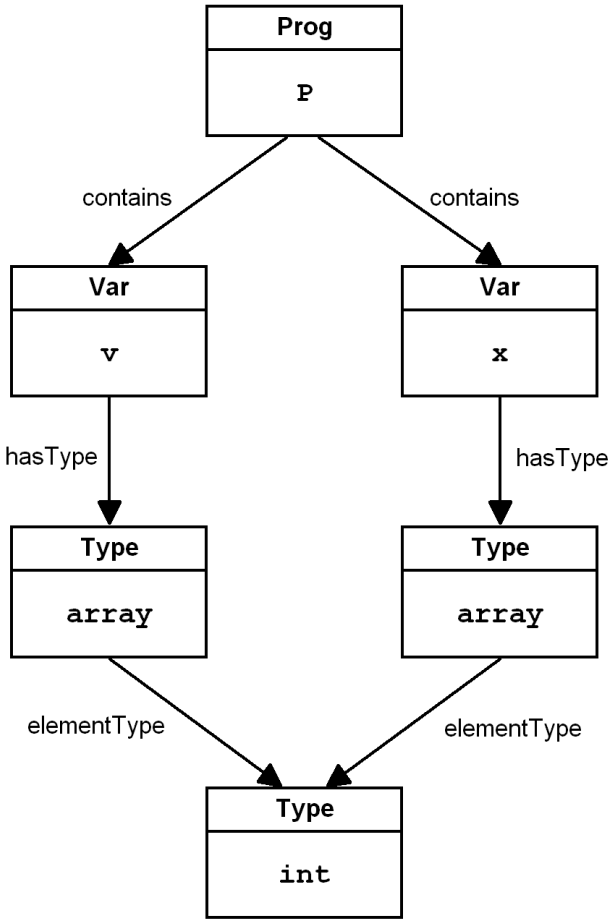

Figure 4. Derived Support for the ’same type’ Concept

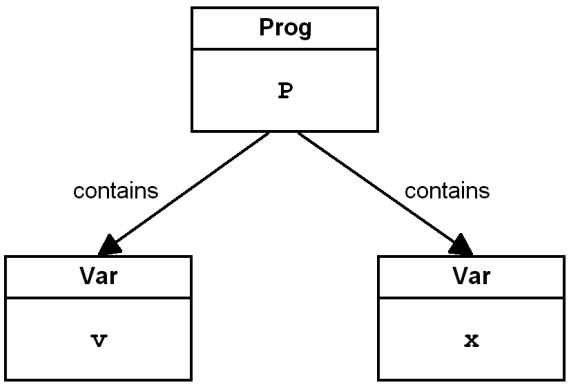

Figure 5. Undefined Support for the ‘same type’ Concept

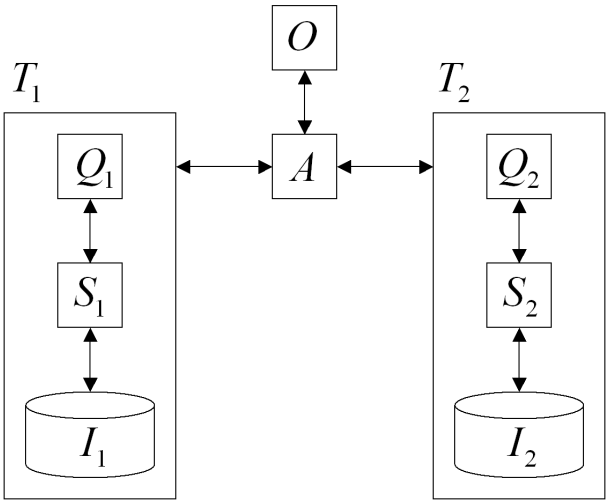

Figure 6. Our Proposed Integration Architecture

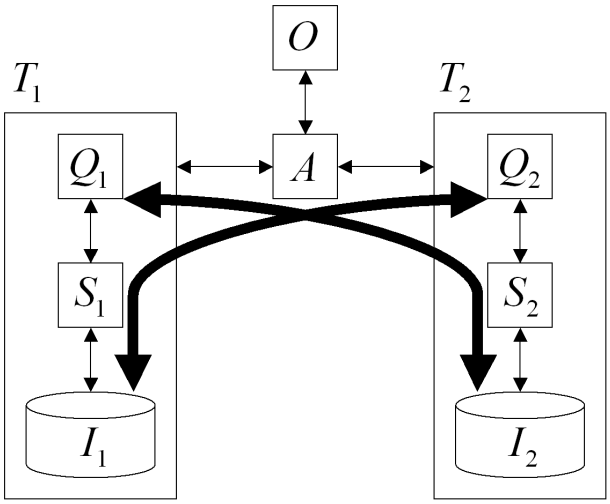

Figure 7. Transparent Interoperability

for either of the variables **v** and **x** is provided, so support for the 'same type' concept is undefined.

We believe that the use of a transaction adapter operating at a conceptual level can facilitate interoperability among reverse engineering tools. In the remainder of this paper we outline our proposal for a conceptual transaction-based approach to integration.

In the context of our discussion, two or more reverse engineering tools want to cooperate in an *integration*. Each of these *participants* has a set of transactions to offer to the group. One or more of these transactions are executed by a tool to carry out a particular *task*. Each task relates in one way or another with one or more concepts supported in the factbase. For example, as we saw above, a type analysis is a task a reverse engineering tool could carry out by querying its factbase for all instances where the 'same type' concept is represented.

8. An Ontological Approach

We start by restricting the domain onto which we will apply our approach to reverse engineering tools. Although this might appear trivial, restricting our domain allows us to fully exploit the similarities among reverse engineering tools (in architecture, graph representation, etc.) that we have discussed so far.

Figure 6 provides a context from which we explain our proposed approach. 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).

Within our restricted reverse engineering domain we can make the following fundamental assumptions:

- There is a significant amount of overlap among the transactions each participant in the integration carries out.
- Although there may be vast differences in the way each factbase is structured, there is a significant amount of information obtainable (natively or derived) from each factbase that is conceptually equivalent.

Capitalizing on these assumptions, we make use of a domain ontology (O) to compile all the conceptual transactions and concepts supported by all participants in the integration. This ontology is essentially a table from which all conceptual transactions and the corresponding concepts that they operate on are recorded. Using this ontology, a *conceptual transaction adapter* (A) translates and filters factbase concepts so they can be used by transactions from other tools.

A key attribute of the conceptual transaction adapter is transparency. The adapter tricks each participant into thinking that the factbase it is accessing is its own. In reality, it is a factbase from one of the other participants in the integration (Figure 7). Neither tool is aware that the adapter is acting as the liaison between them. Successfully implemented, the conceptual transaction adapter would provide seamless, transparent interoperability among all participants in the integration.

9. Building the Ontology

The domain ontology is instrumental in providing the conceptual transaction adapter with the knowledge it needs

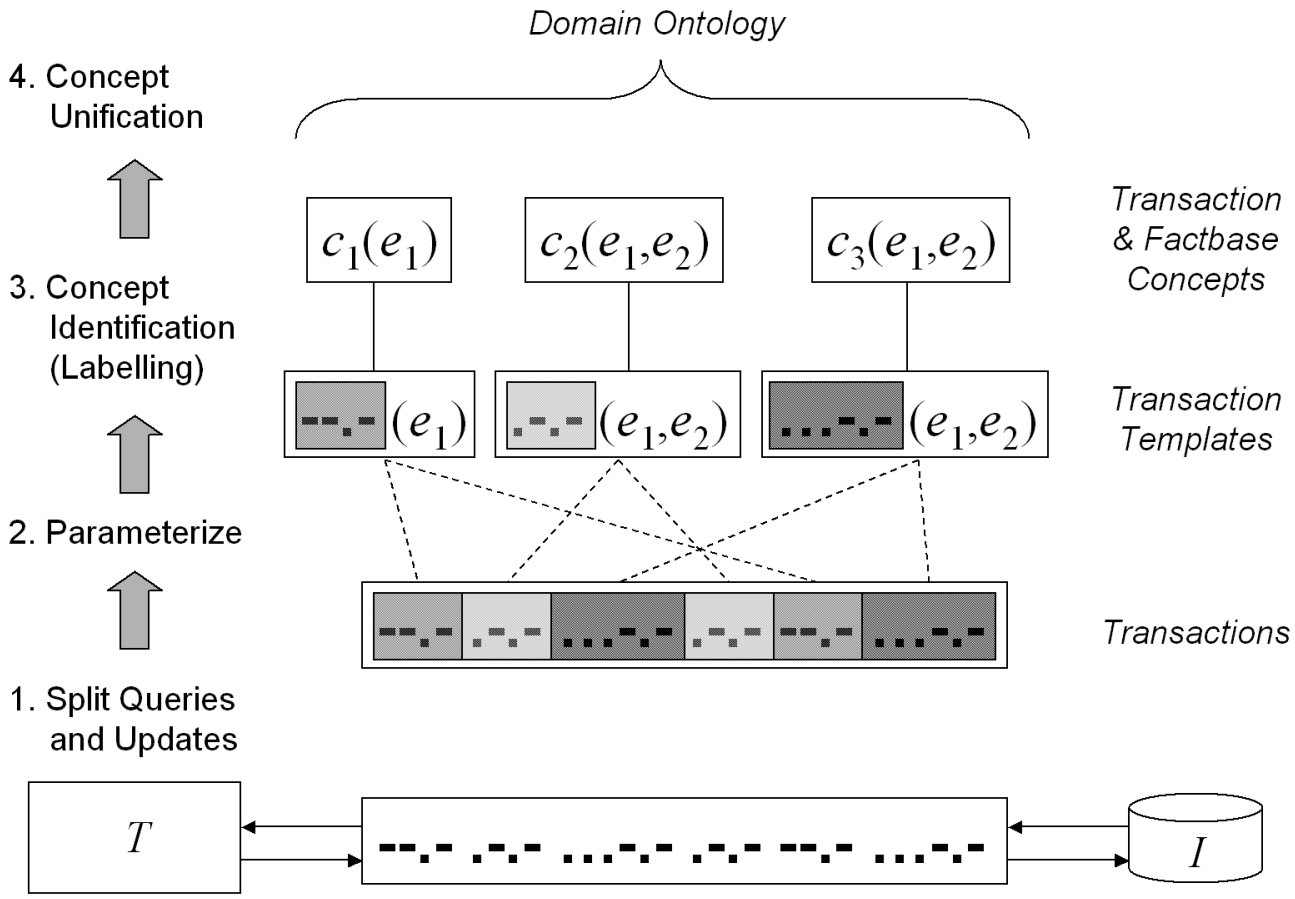

Figure 8. Steps to Building the Domain Ontology

to facilitate interoperability. For this reason it is important that the greatest care be taken to ensure that the ontology created is as comprehensive and complete as possible. We believe the long term benefits of using the ontology far outweighs the short term pain that might be involved in creating it.

Figure 8 shows the steps we propose for a constructive approach to creating the domain ontology. We start at the lowest level of the diagram. In this hypothetical example we see a reverse engineering tool (T) interacting with its factbase (instance I) via queries and updates (respectively shown as dots and dashes). Repeating patterns are readily apparent in the queries and updates the tool performs on its factbase. Each of these patterns is a *transaction* that the tool carries out. In the context of our discussion we use the term transaction in the same sense as it is defined in the database community:

> "an atomic unit of work that is either completed in its entirety or not done at all" [9]

For any given reverse engineering tool a finite set of different kinds of transactions are defined. The first step towards creating the ontology is to split the queries and updates that each tool carries out on its factbase into transactions. Arguably this is the most difficult step because it involves delving into the inner workings (i.e. source code or implementation details) of each tool we want to have participate in the integration. Note that the complete set of transactions identified in this step is the Q we show in Figures 6 and 7.

In the next step we parameterize each of the transactions identified in the first step. Here we are interested in recognizing the entities and relations from the factbase that each transaction works with. These are the concepts represented in each factbase that the transactions operate on. When combined with the information from the first step the result is a set of *transaction templates* for each kind of transaction the tool is capable of performing.

The third step involves applying a nomenclature to the knowledge obtained in each of the previous two steps. So far we have determined the queries and updates that each transaction performs on factbase concepts for each tool in the integration. Now we choose an appropriate label to be

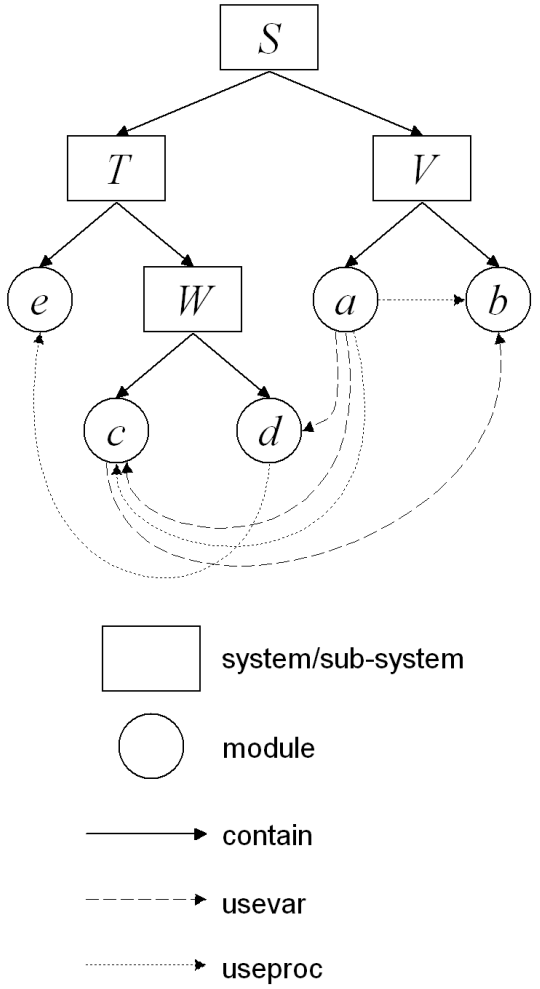

Figure 9. A Graph Representing A Software Architecture

used at the conceptual level for each of the transactions and concepts.

The fourth and final step involves the unification of all the conceptual transactions and factbase concepts into a domain ontology. Recalling the two fundamental assumptions from Section 8, it is likely that a number of conceptual transactions and factbase concepts will overlap. In this step, we eliminate all overlap by combining conceptual transactions and factbase concepts that have the same syntactic or semantic properties.

Completion of these four steps should yield an ontology of conceptual transactions and associated factbase concepts for all the reverse engineering tools participating in the integration.

This methodology for building the domain ontology has a benefit of easily accommodating new participants to the integration as well. Just repeat steps 1 to 3 for any new

```
HLuse :=  (contain +) o
          (useproc + usevar) o
          ((inv contain)+) -
          ID - (contain +) -
          ((inv contain)+)
```

Figure 10. The HLuse Grok Transaction

tool and then complete the 4th step (concept unification) by combining the tool specific results of steps 1 to 3 with the pre-existing domain ontology. Now the new tool should be able to participate in the integration.

10. The Conceptual Transaction Adapter

The conceptual transaction adapter makes extensive use of the domain ontology to get the information it needs to facilitate interoperability among the participants in the integration. It plays a threefold role in the integration. First, *transaction mapping* identifies and dynamically maps actual transactions invoked by a participant tool to the equivalent conceptual transaction in the domain ontology. The ontology reports the factbase concepts that the selected conceptual transaction requires. Next *concept filtering* obtains the required concepts by querying the factbase within the integration that offers the appropriate concept support. The form of the query depends on whether the concept support in the factbase is native or derived. Finally, *syntactic conversion* provides the original actual transaction with the required concepts. These concepts are represented as facts structured according to the schema for the tool the actual transaction originated from.

The example that follows demonstrates how the conceptual transaction adapter would function in the integration.

11. An Example: Sharing the High Level Use Transaction

A very useful aid for maintainers is a view of the architecture of a large software system. One representation of the architecture of a hypothetical software system is shown in Figure 9. This very small entity-relationship diagram (based on an example shown in [11]) features ‘system’, ‘sub-system’ and ‘module’ entities and ‘contain’, ‘usevar’ and ‘useproc’ relations.

Maintainers are often interested in identifying dependencies among the various sub-systems that make up a software system. Looking at Figure 9 it is easy to observe that subsystem W uses sub-system V: module c contained in subsystem W uses a variable from module b contained in subsystem V. In an industrial setting where software systems

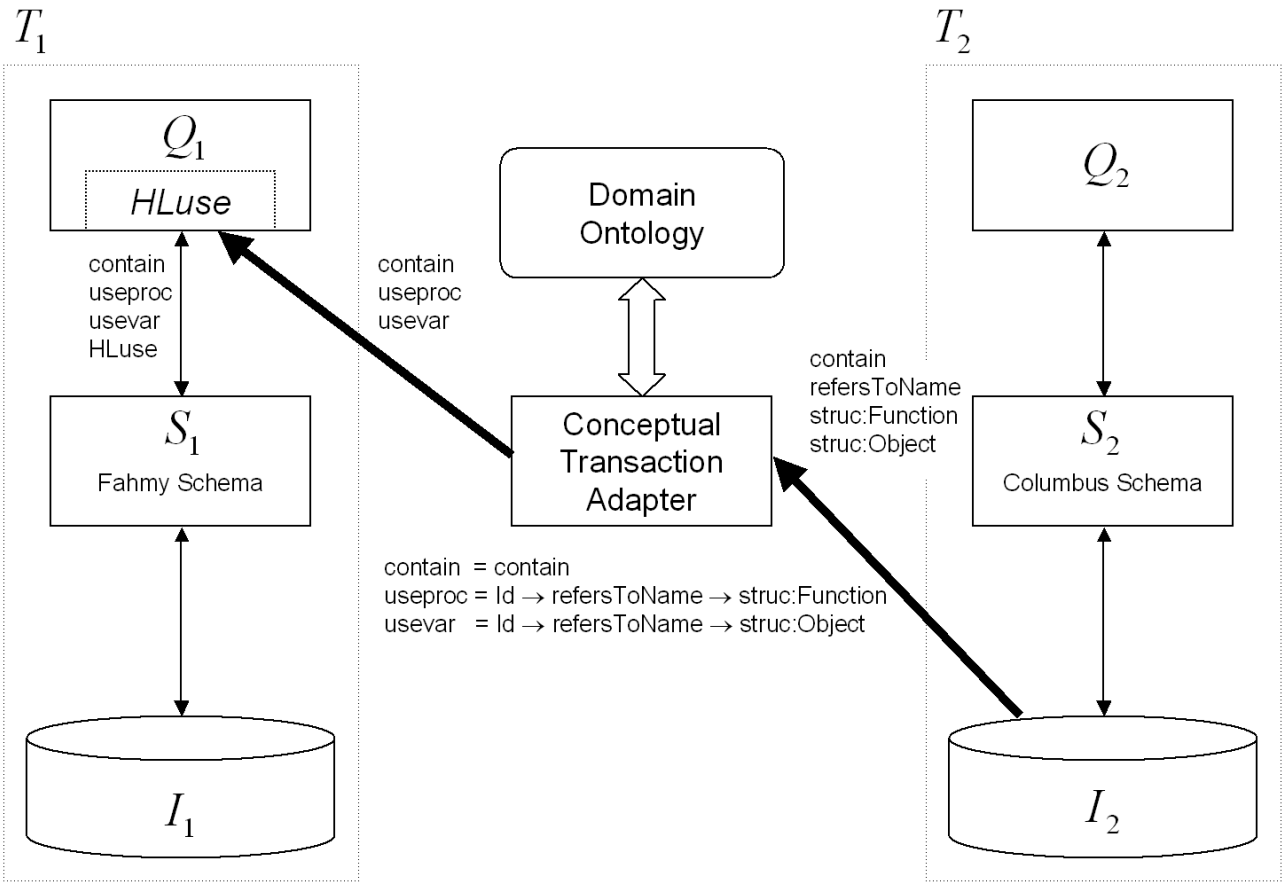

Figure 11. Sharing the HLuse Transaction

are very large, it is not uncommon for a graph that represents a system to be much larger, featuring millions of entities and relations. In such an environment our observation would be impractical if not impossible to make.

Architectural Lifting [17, 12, 28] is an architecture recovery analysis that lifts low-level 'use' relations to higher levels of abstractive detail in the representation of a software system. Using this analysis we can identify dependencies among sub-systems. Fahmy, Holt and Cordy [11] provide *HLuse* (high level use), a transaction that works in the Grok [17, 18] tool to perform architectural lifting on a factbase whose schema supports the representation of the entities and relations shown in Figure 9. The HLuse transaction is shown in Figure 10. The operative details of this transaction are available in [10, 11]. They are beyond the scope of this discussion, so they are not included here. For our purposes, applying this transaction creates a 'HLuse' relationship among two sub-system entities if a 'use' relation among the modules of the two sub-systems exists.

In the example outlined below we describe how the HLuse transaction could be shared among two reverse engineering tools participating in an integration. Figure 11 provides a visual representation of how this could be achieved.

Consider a reverse engineering tool T_1 that implements the HLuse transaction. The tool has a factbase (instance I_1), a schema S_1 that represents (among other entities and relations) the relations 'contain', 'useproc', 'usevar' and 'HLuse'. We call this the Fahmy schema. The tool also has a set of transactions Q_1, one of which is the HLuse transaction.

Now consider a second reverse engineering tool T_2. This tool was built to analyze software representations structured using the Columbus [14] schema. S_2 is the Columbus schema and the factbase I_2 represents software facts structured accordingly. The Columbus schema represents high level semantics and low level structures for source code [13]. It offers native support for the 'contain' concept and derived support for the 'useproc' and 'usevar' concepts. We would like to have T_1 and T_2 participate in an integration with the goal of applying the HLuse transaction to factbase I_2.

We start by building the domain ontology as outlined

in Section 9. The result of this step is a domain ontology 'loaded' with conceptual transactions and factbase concepts from both tools. The conceptual transaction adapter is now ready to facilitate the interoperability we need to achieve our goal.

When the HLuse transaction is invoked from T_1 it is identified by the conceptual transaction adapter. Using the domain ontology the adapter knows it needs 'contain', 'usevar' and 'useproc' concepts from I_2. A query for the natively supported 'contain' concept and derivation queries for the 'usevar' and 'useproc' concepts get the required facts from I_2. This is shown in Figure 11 as the thick line running from I_2 to the adapter. The entity equivalences for the queries are shown under the adapter.

These facts are then syntactically converted so that they are represented in a form consistent with the Fahmy schema. They are then handed over to the HLuse transaction where the architectural lifting analysis is performed. This is shown as the thick line running from the adapter to the HLuse transaction.

The integration facilitated by the conceptual transaction adapter is completely transparent. The HLuse transaction in T_1 performs its analysis on software facts from the I_2 factbase just as though they were facts obtained from the I_1 factbase.

12. Discussion

As we mentioned in Section 9, the process of constructing the domain ontology is by far the most difficult step in our approach to facilitating reverse engineering tool interoperability. Access to the inner workings of a reverse engineering tool, especially a commercial product, may not be possible. Often the unfortunate reality with these systems is that the information contained in the factbase and the services offered are inextricably tied.

On a brighter note, an increasing number of tools provide at least some degree of separation between the factbase they maintain and the services they provide. For instance, tools that import and export information using software exchange languages such as GXL [19], RSF [25] or TA [16] by default separate structural characteristics (i.e. metadata in the form of a schema) from actual data. It is much simpler to construct a domain ontology from these tools as integration participants because the information they represent is explicitly defined. Our approach to integration would work well in this situation because the domain ontology could be used to effectively centralize the mechanisms for sharing information that each tool supports. This would ease maintenance headaches as well. Changes to the representation supported by an integration participant would only need to be reflected in the domain ontology, not in $n(n-1)$ converters as we discussed in Section 4.

13. Conclusions

In this paper we presented a proposal for a novel approach to facilitating transparent interoperability among reverse engineering tools. The primary advantage of our approach to integration is that it does not force any change to preexisting representational structures or operational features (with the exception of transaction piping) of existing reverse engineering tools. All the semantic and syntactic issues related to the software facts maintained by each tool are reconciled during the creation of the domain ontology. All the integrative functionality is built into the conceptual transaction adapter. We believe that our proposed solution would provide transparent interoperability that preserves meaning without loss of detail.

The main disadvantage of our approach is the effort involved in creating the domain ontology. Despite the advantages that come with restricting our domain to reverse engineering tools, this step remains a difficult one to cross. The method we propose involves creating an ontology only for the set of tools participating in the integration. It is hoped that following this method for small groups of tools will lead to the development of best practices that will open the door to optimizations in the ontology creation process. Once created, the use of the ontology to facilitate interoperability is beneficial over the long term for all tools participating in the integration.

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] P. T. Devanbu. "GENOA - A Customizable, front-end retargetable Source Code Analysis Framework". *ACM Transactions on Software Engineering and Methodology*, 9(2), April 1999.

[6] P. T. Devanbu, D. S. Rosenblum, and A. L. Wolf. "Generating Testing and Analysis Tools with Aria". *ACM Transactions on Software Engineering and Methodology*, 5(1):42–62, January 1996.

[7] J. Ebert, B. Kullbach, and A. Winter. "GraX – An Interchange Format for Reengineering Tools". In *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE'99)*, pages 89–98. IEEE Press, 1999.

[8] 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.

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

[10] H. M. Fahmy and R. C. Holt. "Using Graph Rewriting to Specify Software Architectural Transformations". In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'2000)*, Grenoble, France, September 2000.

[11] H. M. Fahmy, R. C. Holt, and J. R. Cordy. "Wins and Losses of Algebraic Transformations of Software Architectures". In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'2001)*, San Diego, California, November 2001.

[12] L. Feijs, R. Krikhaar, and R. V. Ommering. "A Relational Approach to Support Software Architecture Analysis". *Software-Practice and Experience*, 28(4):371–400, April 1998.

[13] R. Ferenc and A. Beszédes. "Data Exchange with the Columbus Schema for C++". In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 59–66, Budapest, Hungary, March 2002.

[14] R. Ferenc, A. Beszédes, M. Tarkiainen, and T. Gyimóthy. "Columbus - Reverse Engineering Tool and Schema for C++". In *Proceedings of the 6th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181, Montréal, Canada, October 2002.

[15] 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. A Position Paper for the ICSE 2000 Workshop on Standard Exchange Formats (WoSEF'00).

[16] R. Holt. An Introduction to TA: The Tuple Attribute Language. Department of Computer Science, University of Waterloo and University of Toronto, November 1998.

[17] R. C. Holt. "Structural Manipulations of Software Architecture Using Tarski Relational Algebra". In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE'98)*, Honolulu, Hawaii, October 1998.

[18] R. C. Holt. Introduction to the Grok Language. `http://plg.uwaterloo.ca/~holt/papers/grok-intro.html`, May 5, 2002.

[19] R. C. Holt and A. Winter. "A Short Introduction to the GXL 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.

[20] 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.

[21] R. C. Holt, A. Winter, A. Schürr, and S. E. Sim. GXL: Toward a standard Exchange Format. Presentation at the 7th Working Conference on Reverse Engineering (WCRE'00), Panel on Reengineering Exchange Formats, November 2000.

[22] R. Kazman, S. G. Woods, and S. J. Carrière. "Requirements for Integrating Software Architecture and Reengineering Models: CORUM II". In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE'98)*, October 1998.

[23] 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`.

[24] T. C. Lethbridge and N. Anquetil. "Architecture of a Source Code Exploration Tool: A Software Engineering Case Study". Technical Report TR-97-07, School of Information Technology and Engineering (SITE), University of Ottawa, November 1997.

[25] J. Martin. RSF File Format. Posted to the Rigi Developer Email Distribution List, August 19 1999.

[26] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong. "Reverse Engineering: A Roadmap". In A. Finkelstein, editor, *The Future of Software Engineering*, International Conference on Software Engineering (ICSE'00), Limerick, Ireland, June 2000. ACM Press.

[27] H. A. Müller, K. Wong, and S. R. Tilley. "Understanding Software Systems Using Reverse Engineering Technology". In *Proceedings of the 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences (ACFAS)*, 1994.

[28] G. C. Murphy, D. Notkin, and K. Sullivan. "Software Reflexion Models: Bridging the Gap Between Source and High-Level Models". In *Proceedings of the Third ACM Symposium on the Foundations of Software Engineering*, Ocotber 1995.

[29] M. Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach.*, volume 1170 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin Heidelberg, 1996.

[30] S. Perelgut. "The Case for a Single Data Exchange Format". In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00)*. IEEE Computer Society Press, November 2000.

[31] C. Riva. "Reverse Architecting: Suggestions of an Exchange Format". In *Proceedings of the Workshop on Standard Exchange Formats (WoSEF) at ICSE'00*, Limerick, Ireland, 2000.

[32] S. E. Sim. "Next Generation Data Interchange: Tool-to-Tool Application Program Interfaces". In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00)*, pages 278–283. IEEE Computer Society Press, November 2000.

[33] 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.