Ontology-Based Software Analysis and Reengineering Tool Integration: The OASIS Service-Sharing Methodology

Dean Jin
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada
djin@cs.umanitoba.ca

James R. Cordy
School of Computing
Queen's University
Kingston, Ontario, Canada
cordy@cs.queensu.ca

Abstract

A common and difficult maintenance activity is the integration of existing software components or tools into a consistent and interoperable whole. One area in which this has proven particularly difficult is in the domain of software analysis and reengineering tools, which have a very poor record of interoperability.

This paper outlines our experience in facilitating tool integration using a service-sharing methodology that employs a domain ontology and specially constructed, external tool adapters. This kind of integration takes advantage of architectural and operational similarities that characterize many of the tools that operate in the software analysis and reengineering domain. A proof of concept implementation among three tools allowed us to explore service-sharing as a viable means for facilitating interoperability among these tools.

1. Introduction

The software reengineering community has actively responded to the needs of maintenance practitioners involved in program comprehension and software analysis. Many tools that provide assistance in carrying out reengineering tasks have been developed. Each of these tools typically provides a specific, specialized functionality to software practitioners [20, 14]. While they are effective operating as independent systems, the usefulness of these tools cannot be maximized without the ability for them to interoperate with other tools [9, 23, 28]. Creation of a suite of tools to support software reengineering requires a means for sharing the services each tool provides among other tools participating in an integration environment.

The *Ontological Adaptive Service-Sharing Integration System (OASIS)* is a novel approach to integration that makes use of specially constructed, external tool adapters and a domain ontology to facilitate software reengineering tool interoperability through service-sharing. This paper provides an outline of the design and function of OASIS. We detail a recently constructed proof-of-concept implementation that enabled interoperability among three reengineering tools working in different programming language domains.

2. Data vs. Service-Sharing Integration

While the motivation to interoperate software reengineering tools remains strong, very little progress towards achieving this goal has been made. Previous approaches to reengineering tool integration have been *data centric*, concentrating on the exchange of data through specialized hard-coded interfaces (APIs) or rigid standardized exchange formats. The main problem with these approaches is that they are *prescriptive*. They force tool developers to provide a particular functionality to another tool or conform to an idiomatic standard in order to participate in the integration process.

For all the time and effort tool developers contribute to the integration process, it generally yields a solution that is specialized for a single tool-to-tool application. Transforming the syntax and semantic information represented in one tool into a form that is compatible with another tool is meticulous and time-consuming. The effort expended in this transformation process is lost when developers want to integrate with additional tools. For n tools that want to interoperate, the transformation process must be repeated $n - 1$ times. As a consequence, data centric integration has left software maintainers with a broad range of autonomous tools that do not work effectively with other tools. Linkages among tools that do interoperate are hard to unravel and difficult to generalize for use in an open integration environment. This makes it difficult for other tool developers to participate in the integration process.

We advocate *non-prescriptive* integration, focusing on sharing the *services* offered by each tool rather than simply exchanging data among them. 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 to software maintainers. In the case of reengineering tools, the inputs are typically source code (or facts about source code) and the output is typically a report or visualization. Often a tool will provide more than one tool service.

Rather than forcing conformance among tools, the OASIS methodology takes advantage of commonalities that exist between them. This is accomplished by focusing on similarities that exist in the representational and operational concepts exhibited by each tool. The result is service-sharing that transcends the representational diversities that exist between tools, enabling interoperability through the application of services based on concepts the tools share.

3. Opportunity in Similarity

Building integration environments among systems that operate in different domains is notoriously difficult. One of the guiding principles of the OASIS methodology is to focus exclusively on tools that operate in a single domain, namely software analysis and reengineering. Doing so permits us to take advantage of three characteristics that many tools in the software analysis and reengineering tool domain share:

1. **Architecture.** Reengineering tools are often constructed and operate in a similar fashion [6]. They commonly consist of three main components: an *information extractor* that parses source code to generate software facts, a *repository* where software facts are stored, and an *analyzer/visualizer* that processes the software facts and reports results to the user [3, 8, 13, 17].

2. **Representation.** Reengineering tools typically represent software facts as a tree or graph structure known, respectively, as an *Abstract Syntax Tree (AST)* or *Abstract Semantic Graph (ASG).*

3. **Operation.** The mechanisms used by reengineering tools to store, manipulate and analyze software facts are very similar to those used by specialized database systems. In this way, most reengineering tools can be characterized as database systems specially tailored to store, manipulate and analyze software facts. Continuing from this perspective, we can abstract the operational characteristics of reengineering tools into the following:

   (a) **Transactions.** The queries and updates that extract, process and analyze the software facts stored in the database. A tool service involves the execution of one or more transactions that yields a result that is relevant to a software maintainer.

   (b) **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 reengineering tools use *Entity-Relationship* (ER) [5] models to define their schemata.

   (c) **Instance.** Software facts stored in the database in a form defined by the schema on which the tool transactions operate. We refer to a reengineering tool database populated with software facts as a *factbase*. The instance for a given reengineering tool is simply the factbase that the tool maintains. A *fact instance* is an individual software fact that has been obtained from the factbase.

The similarities among reengineering tools in terms of architecture, representation and operation provide a significant advantage from an integration perspective. Traditionally, the process of enabling interoperability among tools involves reconciling the differences between various operational paradigms. With reengineering tools these differences are largely nonexistent. This leaves service sharing as the remaining integration challenge to be dealt with.

4 The Role of the Schema

The schema for a tool defines the representation that the tool supports in it's factbase. Since transactions work on a factbase instance, their structure and form are inextricably tied to the schema as well. A change in the schema causes a change in the factbase and a corresponding change in the transactions that operate on the factbase. In this way, the schema acts as a *regulator* for both the factbase content and the transactions that operate on the factbase. The schema also acts as a definitive *reference* for the representational and operational characteristics of a given reengineering tool. It is therefore a key resource for our integration efforts, providing representational and operational details about a reengineering tool.

It is important to note that not all tools that operate in the software analysis and reengineering tool domain have an explicitly defined schema. In many cases the schema is *implicit*. A specific representation exists and transactions operate on it, but no definition or documentation for the schema is provided. In this situation the representation supported by the tool can often be derived from the factbase in a process we call *schema extraction*. This method obtains an estimate of the schema based on the fact instances available in the factbase.

5. The OASIS Architecture

OASIS is an integration methodology that provides a means for reengineering tools to work cooperatively to share services and assist maintainers in carrying out software analysis and program comprehension tasks.

Consider two or more reengineering tools that we want to cooperate in an *integration*. Here we use the term integration to define the environmental boundaries (i.e. the set of tools) that OASIS will operate between. A tool in the integration is referred to as a *participant*.

Each participant offers a set of services to the integration that are shared among the other participants. It is not necessary to share all the services offered by a participant in the integration. At least one service must be shared otherwise there is no justification for the tool to participate. Note that a tool that only supplies a factbase is in fact providing a service, namely one of representing software facts extracted from source code in a particular structured format.

Figure 1 provides an architectural view of OASIS. To keep things simple, we show only two tools (T_1 and T_2) involved in an integration. An actual OASIS implementation can have any number of participants. The makeup of each tool participating in the integration reflects our view of the operational characteristics we mentioned in Section 3. Each of the participant tools (T_1 and T_2) consists of a set of transactions (Q_1 and Q_2), a schema (S_1 and S_2) and a correspondingly structured factbase instance (I_1 and I_2).

Within each tool, a directed, dashed line reflects the important role the schema plays in defining the representation supported by the instance and the structure of the transactions that operate on the instance. A solid, bidirectional line indicates the close operative relationship the transactions have on the instance.

The OASIS methodology involves the creation of two types of components:

1. **Domain Ontology** (O). This component stores all the knowledge required to support service-sharing among each of the tools participating in the integration. The knowledge is stored as a tabularized, cross-referenced compilation of representational concepts and services offered by each integration participant.

   Taken together, the representational concepts stored in the domain ontology define a *conceptual space*, consisting of conceptual 'slots' that fact instances fit into. A fact instance fits into a slot only when the concept it represents matches a concept in the domain ontology. We say that a tool has *concept support* when this occurs. We describe concept support in more detail in an earlier paper [18]. Shared services only operate on fact instances that actually fit into these conceptual slots.

   A service offered by a tool participating in an OASIS integration can be shared only when the concepts required by the service intersect with the concepts supported by another participant tool.

   Each OASIS implementation requires only one domain ontology.

2. **Conceptual Service Adapters** (A_1, A_2). These components function as integration facilitators for tools participating in the integration. In an OASIS implementation, each tool is affiliated with a single conceptual service adapter. Each makes use of the domain ontology to get the information it needs to regulate the integration process.

   Conceptual service adapters perform the following three main functions:

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

   (b) *Factbase Filtering.* Depending on the mode of operation invoked, each conceptual service adapter will map all fact instances into and out of the conceptual space defined by the domain ontology. This process is known as *filtering*. Mapping fact instances into the conceptual space is performed by an *inFilter*. Mapping from the conceptual space is performed by an *outFilter*. Both of these filters are specially tailored to work with the representation supported by the factbase for the tool that the conceptual service adapter is associated with.

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

   Although all the conceptual service adapters have the same basic architecture and operating characteristics, each is specially constructed to handle the functional and information filtering aspects of it's corresponding tool that are required to facilitate interoperability.

The communication links between the domain ontology, the conceptual service adapters, and tools they are associated with are shown as solid black lines in Figure 1.

6 How OASIS Works

In order to show how an OASIS implementation works, consider the two reengineering tools T_1 and T_2 as shown in

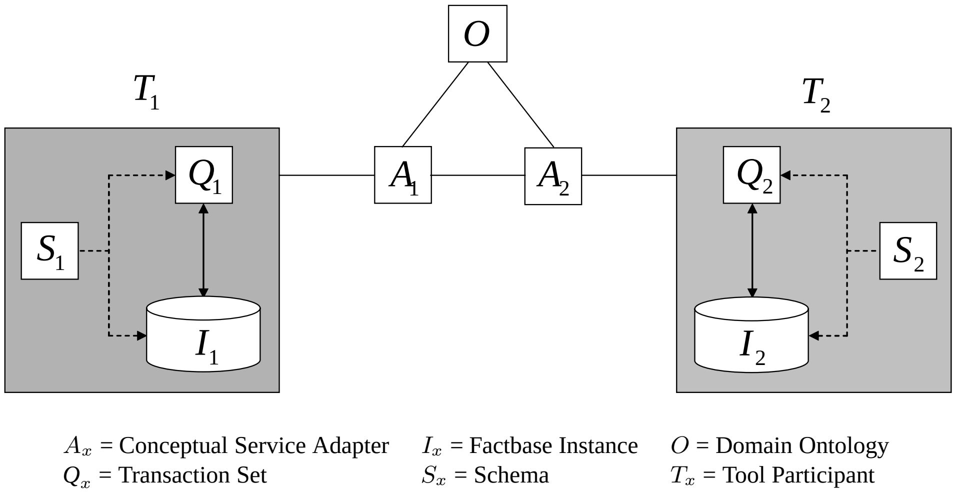

Figure 1. The OASIS Architecture

Each tool (T_x) consists of a factbase *instance* (I_x) whose form is dictated by a *schema* (S_x). A set of transactions (Q_x) conform to the schema and operate on the instance. OASIS makes use of a *domain ontology* (O) and tool-specific *conceptual service adapters* (A_x) to facilitate service-sharing among the tools participating in the integration.

Figure 1. This is the base case for our integration paradigm. An OASIS implementation can have any number of participants. We only show two here to keep the explanation on how OASIS works as simple as possible.

The goal of our integration effort is to apply a service available in one participant to the factbase of another participant. In this example, T_2 offers a service V consisting of the sequential application of transactions q_x, q_y and q_z (a subset of the complete set of transactions offered by Q_2). We would like to apply service V to I_1, the factbase for T_1. This has the effect of sharing service V with T_1.

The domain ontology O has been constructed and contains knowledge of the representational concepts and services supported by tools T_1 and T_2. The conceptual service adapters A_1 and A_2 facilitate the interoperability we need to achieve our goal. Although a number of steps are involved in the integration process, the function of OASIS can be characterized in five phases. Each of these phases is discussed in more detail below.

Phase 1 - Service Request and Identification

This phase involves the tool requesting the service (T_1), it's corresponding conceptual service adapter (A_1) and the domain ontology (O). A request for service V invoked from T_1 is received by A_1. The adapter uses the ontology to identify V as a service offered by T_2. It also learns that V requires a factbase that supports (in this example) three concepts known as c_1, c_2 and c_3 in the ontology.

Phase 2 - Checking for Concept Support

This phase involves the conceptual service adapter A_1 and the ontology. The factbase for T_1 must support the concepts that service V operates on. The ontology is used to identify the concepts represented by the tool calling the service to ensure that the proper concept support exists. In our example, A_1 accesses O for a second time and verifies that T_1 supports a representation for concepts c_1, c_2 and c_3. If a tool does not support the required concept(s), the integration attempt will terminate at this point.

Phase 3 - Providing the Facts

A_1 invokes it's inFilter to map all fact instances from the I_1 factbase into the conceptual space. A_1 then sends a request to A_2 asking it to execute service V, providing a pointer to the conceptual space.

Phase 4 - Service Execution

Acting on the request from A_1, A_2 invokes it's outFilter to map the conceptual space representation to a local factbase instance for T_2. A_2 then instructs T_2 to apply service V to the instance. After service V is completed, A_2 invokes it's inFilter to map the results of the service to the conceptual space. It then sends a message to A_1 indicating the service has completed and provides a pointer to the conceptual space.

Phase 5 - Returning the Results
Acting on the message from A_2, A_1 invokes it's outFilter to map the conceptual space representation to a local factbase for T_1. The completed integration terminates.

In this example, service V is essentially *shared*; it can be applied to fact instances from I_1 and I_2. Any reengineering tool that supports concepts c_1, c_2 and c_3 can share service V from T_2 in this manner.

Using Figure 1 as a reference, the effect of service-sharing can be indicated by a solid bidirectional line stemming from a set of transactions from one tool (Q_x) to a factbase instance of another tool (I_y).

7. Proof of Concept Implementation

The goal of our proof-of-concept implementation was to demonstrate the feasibility of OASIS through the development of a functional integration among a set of reengineering tools. The following steps based on our development experiences provide useful a guideline for the implementation process:

1. **Tool Requirements**. A tool must exhibit a number of characteristics in order for it to be successfully brought into an OASIS implementation. For example, in relation to *accessibility*, a tool must store fact instances in a central location that is accessible to the conceptual service adapters. Tools must also have *definable service transactions* and a clear separation of fact instances from the transactions that operate on them (*service-factbase separation*). In this stage, an assessment of a candidate tool in relation to these and other requirements is made.

2. **Ontology Development/Augmentation**. This step involves identifying and organizing into a domain ontology all the representational and service related concepts for each of the tools participating in the integration. When a new tool is introduced to the OASIS implementation, additional representational and service concepts are added to the ontology.

3. **Conceptual Service Adapter Construction**. One conceptual service adapter for each tool participant is created. Each adapter manages all aspects of the integration as it relates to it's corresponding tool.

4. **Testing and Incorporation**. A comprehensive set of tests are performed. All components that were created to enable the tool to participate in the integration are tested individually. This is followed by system testing, where the tool is brought into the existing OASIS implementation and tested online with other tools. If no problems are identified throughout the testing process, then the tool is considered fully incorporated into the OASIS implementation.

8. Participant Tools

Three tools were chosen to participate in our OASIS proof of concept implementation:

Advanced Software Design Technology (ASDT)

ASDT was developed in 1991 as part of a collaboration among researchers in the Software Technology Lab at Queen's University and the IBM Center for Advanced Studies (CAS) in Toronto [19, 7]. ASDT provides design recovery analysis of source code written exclusively in *Turing Plus* [15, 16].

ASDT has two main operational phases. In the *recovery* phase, a Turing Plus source code artifact is input and a factbase consisting of raw design facts expressed in a proprietary Prolog-like notation is produced. In the *exploration* phase the user obtains detailed information about an entity and its relationship to other entities that exist in the factbase. ASDT provides two tool services that are of interest to our integration efforts:

- *Query*. Given the name of an entity in the system, the service outputs all relevant facts about the entity. Relevant facts include direct and indirect relationships of the entity to others in the system. Indirect relationships are synthesized through transitive closure on relations associated with the entity.

- *Slice*. Given the name of an entity in the system, the service produces a file that contains all fact instances sliced from the direct and indirect relationships identified in the query service

Fahmy Tool

Hoda Fahmy and colleagues have extensively explored the use of graph transformations to support maintenance tasks related to software architectures. The *Fahmy Tool* implements three of the graph transformations defined in [10] in a tool executed from the Linux command line:

- *High Level Use*. An architecture recovery analysis that lifts low-level use relations to higher levels of abstractive detail in the representation of a software system.

- *Hide Interior*. This service collapses the details of a selected subsystem, hiding it's interior components.

	Reengineering Tool		
	ASDT	*Fahmy Tool*	*Rigi*
Programming Language Domain	Turing+	PLIX C	-
Schema Characteristics	10 Entities 13 Relationships 98 Constraints	3 Entities 4 Relationships 10 Constraints	Graph schema defined by user in proprietary domain files.
Factbase Syntax	Proprietary (Prolog-like)	RSF	RSF
Services Offered	Query Slice	High Level Use Hide Interior Hide Exterior	Visualization Spring Layout Sugiyama Layout

Figure 2. Characteristics of the OASIS Implementation Participants

Relationships among components in the subsystem to/from external entities are preserved.

- *Hide Exterior*. This service focuses exclusively on a selected subsystem, hiding all exterior components. External relationships are preserved as links to unknown entities.

Rigi

Rigi [21, 1] is a reengineering tool that is the product of over ten years of research and development at the University of Victoria. Our primary reason for including Rigi in our OASIS implementation was to take advantage of the following services:

- *Visualization*. This service provides the user with a graphical view of a factbase provided. Entities are displayed as square nodes and relationships are shown as arcs (lines) connecting two nodes together. Once the graph has been loaded into Rigi, the user can manipulate it and invoke any of the layout options that Rigi provides.

- *Sugiyama Layout* [27]. This is a pre-configured graph manipulation procedure that arranges the nodes in the graph into a hierarchical, tree-like form that reduces the crossing of arcs as much as possible.

- *Spring Layout* [12]. This is a pre-configured graph manipulation procedure that arranges the nodes of a graph based on a measure of the connectedness that each node has with other nodes. Highly connected nodes are arranged closer together, while nodes with low connectivity are arranged further apart. In this way, the relationship arcs that connect graph nodes together act like ‘springs’.

Table 2 summarizes the characteristics of each of the tools chosen to participate in our OASIS proof-of-concept implementation. For each tool the following aspects are shown:

- **Programming Language Domain**. This is the programming language(s) supported by the tool parser.

- **Schema Characteristics**. Statistics for the schemas are stated in terms of the entities, relationships and constraints that exist in the representation.

- **Factbase Syntax**. This is the structural characteristics of the factbase supported by each tool. The syntax of the factbase is independent of the information the facts convey.

- **Services Offered**. These are the services that each tool provides that we share in the OASIS implementation.

Table 2 provides an indicator of the diversity inherent in the tools that make up our integration participants. Three different programming language domains are represented: Turing+, PLIX and C. The representation for ASDT is highly constrained both in the native schema and in the representations produced by the services. In contrast, Fahmy Tool has a smaller schema and has many fewer constraints. ASDT’s Prolog-like factbase syntax is completely different from RSF used in Rigi and Fahmy Tool.

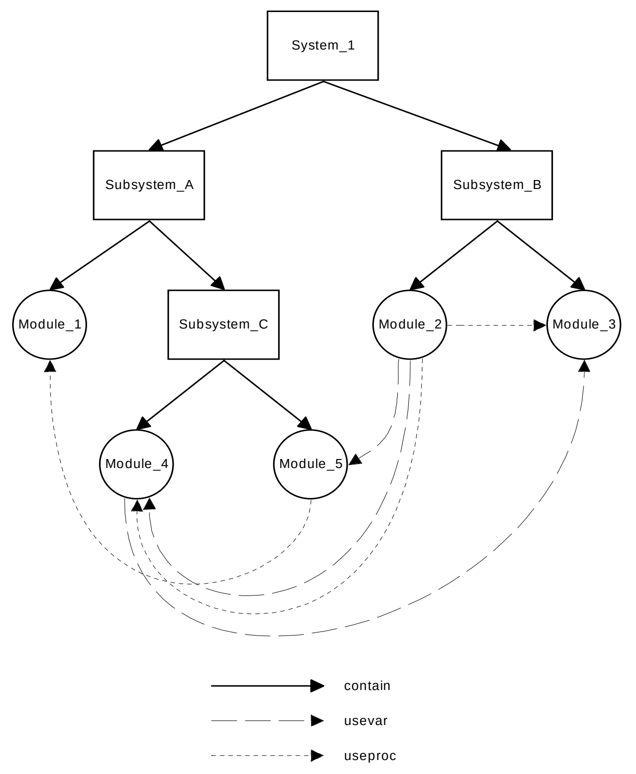

Figure 3. A Software Architecture Example

```
  $INSTANCE System_1 system
  $INSTANCE Subsystem_A subsystem
  $INSTANCE Subsystem_B subsystem
  $INSTANCE Subsystem_B subsystem
  $INSTANCE Module_1 module
  $INSTANCE Module_2 module
  $INSTANCE Module_3 module
  $INSTANCE Module_4 module
  $INSTANCE Module_5 module
  contain System_1 Subsystem_A
  contain System_1 Subsystem_B
  contain Subsystem_A Module_1
  contain Subsystem_A Subsystem_C
  contain Subsystem_C Module_4
  contain Subsystem_C Module_5
  contain Subsystem_B Module_2
  contain Subsystem_B Module_3
* usevar Module_2 Module_5
  usevar Module_2 Module_4
  usevar Module_4 Module_3
  useproc Module_2 Module_3
  useproc Module_2 Module_4
  useproc Module_5 Module_1
```

Figure 4. RSF facts for Figure 3 Example

9. A Step-By-Step Example: Sharing the ASDT Query Service

The architecture of a tiny hypothetical software system often used to demonstrate Fahmy Tool is shown in Figure 3. Looking at this figure it is easy to observe, for example, that `Module_5` uses a procedure in `Module_1`. In an industrial setting it is not uncommon for a graph representing the architecture of a system to be much larger, featuring thousands of entities and relations. In such an environment our observation would be impractical if not impossible to make.

Maintainers are often interested in focusing on a particular part of a system to determine how it is interconnected with other components. Fahmy Tool does not have a service that fulfills this requirement (the Hide Exterior service does not identify destination entities in external relations). In this example, we use the Query service shared by ASDT to obtain information about the `Module_5` entity. The flow of information through the OASIS components is shown for each of the following steps in Figure 8.

❶ We start with the Fahmy Tool factbase stored in RSF shown in Figure 4. The user calls the conceptual service adapter (CSA) for Fahmy Tool and requests the Query service.

❷ The RSF file is mapped to the conceptual space by the Fahmy Tool CSA inFilter and a message is sent to the ASDT CSA requesting the Query service.

❸ The ASDT CSA receives the message and uses its outFilter to map facts from the conceptual space to an ASDT structured factbase file. This is shown in Figure 5. Note that extra facts originally produced by the Fahmy Tool inFilter are noticeable here. In particular, the `usevar` relationship highlighted with a '*' in Figure 4 is mapped to *three* facts highlighted with a '*' in Figure 5. This preserves the semantics of Fahmy Tool's `usevar` relationship in both the conceptual space and the ASDT factbase subsequently obtained from the conceptual space.

❹ The Query service is invoked by the ASDT CSA which produces results in a new ASDT factbase shown in Figure 6.

❺ The ASDT CSA uses its inFilter to return the results through the conceptual space. At the same time it sends a message to the Fahmy Tool CSA that the results are ready.

❻ The Fahmy Tool CSA receives this message and uses its outFilter to map from the conceptual space to the RSF file shown in Figure 7.

```
  program(System_1).
  module(Subsystem_A).
  module(Subsystem_B).
  module(Subsystem_C).
  procedure(Module_1).
  procedure(Module_2).
  procedure(Module_3).
  procedure(Module_4).
  procedure(Module_5).
  variable(variableIn_Module_4_1).
  variable(variableIn_Module_5_1).
* variable(variableIn_Module_3_1).
  contains(System_1,Subsystem_A).
  contains(System_1,Subsystem_B).
  contains(Subsystem_A,Subsystem_C).
  contains(Subsystem_A,Module_1).
  contains(Subsystem_B,Module_2).
  contains(Subsystem_B,Module_3).
  contains(Subsystem_C,Module_4).
  contains(Subsystem_C,Module_5).
  contains(Module_3,variableIn_Module_3_1).
  contains(Module_4,variableIn_Module_4_1).
* contains(Module_5,variableIn_Module_5_1).
  calls(Module_2,Module_3).
  calls(Module_2,Module_4).
  calls(Module_5,Module_1).
  read_ref(Module_2,variableIn_Module_4_1).
* read_ref(Module_2,variableIn_Module_5_1).
  read_ref(Module_4,variableIn_Module_3_1).
```

Figure 5. ASDT Input from Fahmy Tool

```
procedure(Module_5).
contains(Subsystem_C,Module_5).
contains(Module_5,variableIn_Module_5_1).
calls(Module_5,Module_1).
```

Figure 6. ASDT Query Service Results

```
$INSTANCE Module_5 module
contain Subsystem_C Module_5
contain Module_5 variableIn_Module_5_1
useproc Module_5 Module_1
```

Figure 7. RSF Results from Query Service

This tiny example serves to demonstrate the technique. In practice we have demonstrated integration between all of the shared services offered by all three of the tools in our OASIS integration. We have applied the system towards the analysis of production-sized software systems including Linux Kernel v2.0.27a (14,338 facts), the TXL language processor v6.0 (9,000 Turing+ LOC, 6,780 design facts) and IBM's Tobey code generation system (250,000 PLIX LOC, 11,066 architecture facts).

10. Discussion

While OASIS is a new methodology, we would argue that it represents the next logical step in the development of integration technologies for the reengineering tool community. Integration solutions have progressed from very rigid, proprietary formulas to more flexible, increasingly generalized approaches as they become more useful to software practitioners.

One of the major pitfalls of previous attempts to facilitate integration among reengineering tools has been the prescriptive methodologies that tool developers have been forced to work with. The primary goal of OASIS is to simplify the work involved in participating in the integration process. We believe that this can be accomplished by maintaining a clear separation between each participant and the components that look after the complexities of integration.

OASIS provides a conceptual interface (the domain ontology) and operational interfaces (conceptual transaction adapters) to each participant that facilitate integration without revealing the details of their implementation.

11. Related Work

Reverse engineering tool integration is a difficult and important problem that has been studied by many groups using many different methods. In the Software Bookshelf [11], a central repository design is used in which entities and relationships are represented as "facts" in the tuple-attribute notation TA. The Software Bookshelf integrates tools that subscribe to the TA notation but (intentionally) does not provide any fixed semantics or schema for the facts, other than the single shared "contains" fact (although in practice the "uses" fact is often also shared). The bookshelf provides a very general framework for integration of tools that subscribe to the TA notation and a shared set of agreed facts. It does not address conceptual semantics and provides no assistance for integrating tools using other data representations, schemas or languages.

An earlier related experiment [4] used the Telos knowledge representation language [22] to implement a common repository at a conceptual level describing an object-oriented shared global schema in some ways similar to our

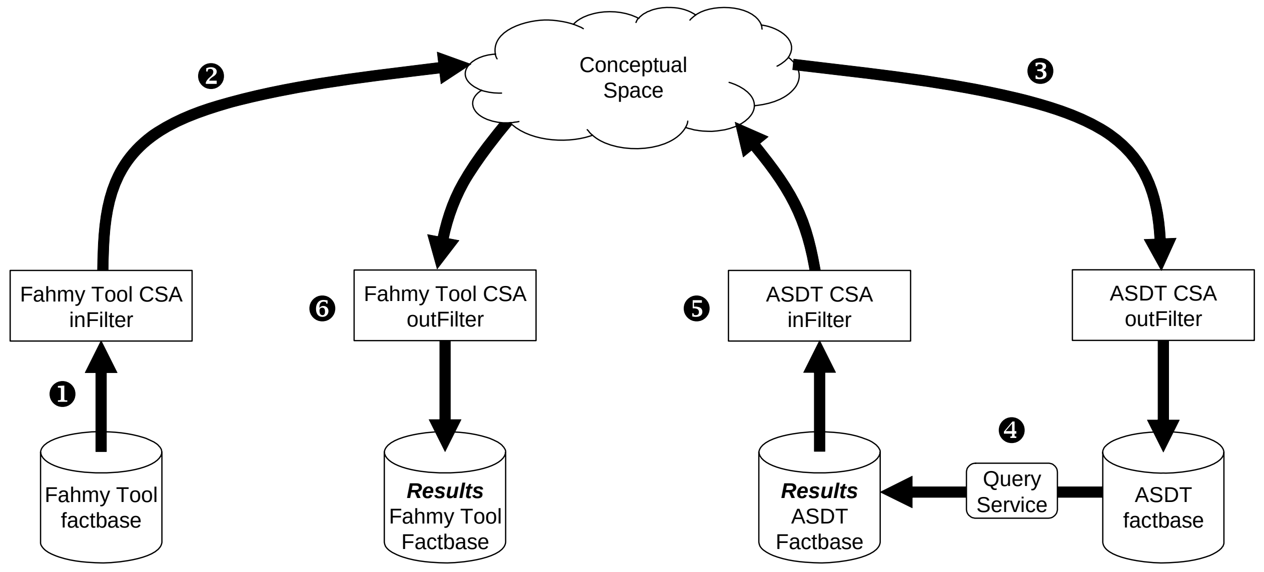

Figure 8. Sharing the ASDT Query Service

inferred example ontology. The project involved an experiment with a single software system and a fixed set of tools, using hand adaptation to the shared database and schema. The net effect for the experimental system was similar to our results in that it matched a set of mixed tools to work together, and very similar in the sense that individual tools could see different slices or views of the information about the system. However, no attempt was made to generalize the results to other systems, concepts or tools and most of the work was done by hand.

The integration of software engineering tools in general has a long history with many interesting proposed solutions. One of the best known is the work of Stephen Reiss on GARDEN [24] and later FIELD [25]. Each of these is a comprehensive programming environment designed for integrating a range of tools, including both language-dependent custom tools and more general Unix utilities, into a single consistent interface for program development. Integration uses the idea of a central message server that allows tools to send messages to one another via “wrappers” corresponding roughly to our conceptual adaptors. Messages were attached to software source and other artifacts using “annotations” whose meaning was agreed to by participating tools. While this work was very successful at providing a framework for a consistent human interface to host a range of tools, it did not provide any constructive help in building the shared understanding or adaptation of the tools involved.

Perhaps the granddaddy of all software engineering tool integration efforts is the IDL work of Snodgrass and Shannon [26]. Using IDL, the actual representation of data and schema is hidden behind a high level interface description that provides a consistent language-independent API for accessing shared knowledge. Integration of tools is done by programming each to subscribe to the shared interface, and an automated process links tools implemented using different language and technologies together.

Our work differs from all these efforts in that we provide a constructive method for building a shared conceptual ontology from the observed concepts provided or used by the tools to be integrated, independent of language and technology, and a non-invasive method for integrating the tools to the ontology using conceptual adaptors that do not require any reprogramming of the tools themselves. In essence, using our method the tools to be integrated can be off-the-shelf black boxes, and thus our work is well suited to the “COTS” integration problem [2] for reverse engineering components. Our work is also independent of schema, technology, environment or system. Like IDL, it provides a general method for integration independent of the nature of the tools, but unlike IDL, it does not require reprogramming to use a shared API.

12 Conclusion

This paper outlined the *Ontological Adaptive Service-Sharing Integration System (OASIS)*, a new, an ontology-based shared-service integration paradigm for reengineering tools. To demonstrate the effectiveness of our design we completed a proof of concept implementation among three tools. Using the system implemented, each tool can invoke a shared-service that operates on its own factbase to carry out an analysis offered by another tool operating independently in the system.

References

[1] Rigi Group Home Page. URL: `http://www.rigi.csc.uvic.ca/`.

[2] B. W. Boehm and C. Abts. “COTS Integration: Plug and Pray?”. *IEEE Computer*, 32(1):135–138, 1999.

[3] 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.

[4] E. Buss, R. D. Mori, W. M. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, H. A. M. E. Merlo AND, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S. R. Tilley, J. Troster, and K. Wong. “Investigating reverse engineering technologies for the CAS program understanding project.”. *IBM Systems Journal*, 33(3):477–500, 1994.

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

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

[7] J. R. Cordy and K. A. Schneider. “Architectural Design Recovery Using Source Transformation”. In *CASE’95 Workshop on Software Architecture*, Toronto, July 1995.

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

[9] 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, 1999.

[10] 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.

[11] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. “The Software Bookshelf”. *IBM Systems Journal*, 36(4):564–593, November 1997.

[12] T. Fruchtermann and E. Reingold. “Graph drawing by force-directed placement”. Technical Report UIUC CDS-R-90-1609, Department of Computer Science, University of Illinois at Urbana-Champaign, 1990.

[13] 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.

[14] G. Y. Guo, J. M. Atlee, and R. Kazman. “A Software Architecture Reconstruction Method”. In P. Donohoe, editor, *Software Architecture*, TC1 1st Working IFIP Conference on Software Architecture (WICSA 1), pages 15–33, San Antonio, Texas, February 1999. Kluwer Academic Publisher.

[15] R. C. Holt and J. R. Cordy. The Turing Plus Report. Computer Systems Research Group, University of Toronto, February 1987.

[16] R. C. Holt and J. R. Cordy. “The Turing Programming Language”. *Communications of the ACM*, 31(12), December 1988.

[17] 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.

[18] 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.

[19] D. A. Lamb and K. A. Schneider. “Formalization of Information Hiding Design Methods”. In J. Botsford, A. Ryman, J. Slonim, and D. Taylor, editors, *Proceedings of the 1992 Centre for Advanced Studies Conference (CASCON’92)*, pages 201–214, Toronto, Ontairo, November 1992.

[20] T. C. Lethbridge. Requirements and Proposal for a Software Information Exchange Format (SIEF) Standard. Draft Manuscript, November 21 1998. URL: `http://www.site.uottawa.ca/~tcl/papers/sief/standardProposal.html`.

[21] H. A. Müller and K. Klashinsky. “Rigi – A system for Programming-in-the-Large”. In *Proceedings of the International Conference on Software Engineering (ICSE’88)*, pages 80–86, 1988.

[22] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. “Telos: Representing Knowledge About Information Systems.”. *ACM Transactions on Information Systems*, 8(4):325–362, 1990.

[23] 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.

[24] S. P. Reiss. “GARDEN Tools: Support for Graphical Programming”. In *Proceedings of the International Workshop on Advanced Programming Environments*, pages 59–72, Trondheim, Norway, 1986.

[25] S. P. Reiss. *FIELD: The Friendly Integrated Environment for Learning and Development*. Kluwer Press, 1994.

[26] R. Snodgrass and K. Shannon. “Supporting flexible and efficient tool integration”. In *Proceedings of the International Workshop on Advanced Programming Environments*, pages 290–313, Trondheim, Norway, 1986.

[27] K. Sugiyama, S. Tagawa, and M. Toda. “Methods for visual understanding of hierarchical systems”. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(4):109–125, 1981.

[28] 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, Ischia, Italy, June 1998.