Integrating Reverse Engineering Tools Using a Service-Sharing Methodology

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

James R. Cordy
School of Computing
Queen's University, Kingston, 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 re-engineering tools, which have a relatively poor record of interoperability. This paper outlines our experience in facilitating interoperability between three such tools using OASIS, a service-sharing methodology that employs a domain ontology and specially constructed, noninvasive tool adapters.

1. Introduction

The program comprehension community has responded to the needs of practitioners involved in software maintenance with many tools to provide assistance in reverse- and re-engineering tasks. Typically each of these provides a specific, specialized functionality [9, 17]. While they can be effective as independent systems, the usefulness of these tools can be limited by their inability to interoperate with other tools [6, 20, 24]. Creation of a suite of tools to support software analysis and re-engineering requires a means for sharing the services each tool provides with the other tools in an integration environment.

In the *Ontological Adaptive Service-sharing Integration System (OASIS)* [15] we have proposed a novel, non-invasive approach to integration that uses specially constructed tool adapters and a domain ontology to facilitate tool interoperability through service sharing. This paper presents a first experiment using OASIS to facilitate integration of three diverse reverse engineering tools normally aimed at quite different languages and tasks.

2. The OASIS Architecture

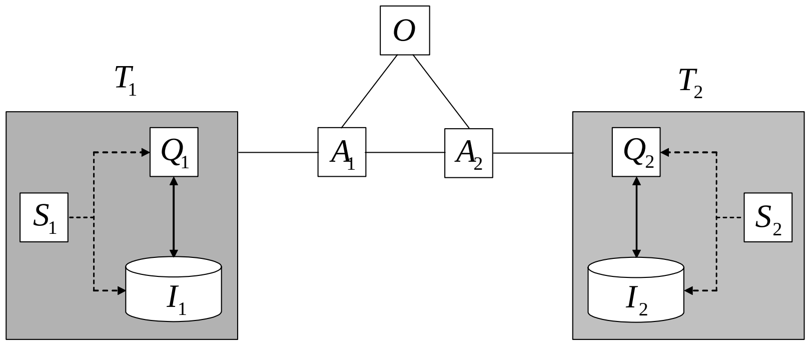
OASIS provides a means for tools to work cooperatively to share services and assist maintainers in carrying out software analysis and program comprehension tasks. Consider two or more analysis or re-engineering tools that we want to cooperate in an integration. We use the term *integration* to refer to 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* that are shared with the other participants. Note that even a tool that simply supplies a factbase provides such a service, namely the extraction of facts from source code.

Figure 1 shows an architectural view of OASIS. Although an OASIS implementation may have any number of participants, for simplicity we show only two tools in this integration. The operational characteristics of each of the participant tools (T_1 and T_2) is characterized by a set of transactions (Q_1 and Q_2), a schema (S_1 and S_2) and a corresponding structured factbase instance (I_1 and I_2). The dashed line inside each tool reflects the important role the schema plays in defining the representation of the instance and the structure of the transactions that operate on it. A solid, bidirectional line indicates the close relationship between the transactions and the instance.

The OASIS methodology involves the creation of two integration components: a *domain ontology*, representing the shared conceptual space of the tools, and a set of *conceptual service adapters* adapting each of the tools to it.

Domain Ontology (O). The domain ontology stores the knowledge required to support service sharing between the tools as a tabularized, cross-referenced compilation of the representational concepts and services offered by each integration participant. Together, the representational concepts define a *conceptual space* consisting of a set of conceptual 'slots' that fact instances may 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 discuss concept support in more detail in a previous paper [13]. Shared services only operate on fact instances that can fit into these conceptual slots. A service offered by a tool participating in an OASIS integration can be shared only when the concepts it requires intersect with the concepts supported by the other tool.

Conceptual Service Adapters (A_1,A_2). Conceptual service adapters act as integration facilitators for participating tools. Each tool is associated with a single conceptual service adapter that uses the domain ontology to provide the information needed to regulate the integration process. Conceptual service adapters perform three main functions:



A_x = Conceptual Service Adapter I_x = Factbase Instance O = Domain Ontology
Q_x = Transaction Set S_x = Schema T_x = Tool Participant

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 uses a domain ontology (O) and tool-specific conceptual service adapters (A_x) to facilitate service sharing among the tools participating in the integration.

(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, conceptual service adapters map fact instances into and out of the conceptual space defined by the domain ontology. This process is known as *filtering* [14]. Mapping fact instances into the conceptual space is performed by an *inFilter*, and mapping from the conceptual space is performed by an *outFilter*. Both filters are tailored to work with the factbase representation and semantics of the particular tool 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 of the tool they are associated with.

Although all conceptual service adapters have the same basic architecture and operating characteristics, each is specially tailored to handle the functional and information filtering aspects required to facilitate interoperability with its corresponding tool. The access and communication links between the domain ontology, the conceptual service adapters and the tools they are associated with are shown as solid black lines in Figure 1.

3. Proof of Concept Experiment

The goal of our experiment was to demonstrate the feasibility of OASIS through development of a functional integration of a small set of diverse software analysis tools. The following steps outline our implementation process:

1. **Tool Requirements Analysis**. A tool must exhibit a number of characteristics in order to be successfully brought into an OASIS implementation. In relation to *accessibility*, a tool must store fact instances in a way that is accessible to the conceptual service adapters. Tools must 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 requirements is made.
2. **Ontology Development and Augmentation**. This step involves identifying and organizing into a domain ontology all the representational and service related concepts for the tools participating in the integration. When a new tool is integrated, new 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 its corresponding tool.
4. **Testing and Incorporation**. All components created to enable the tool to participate in the integration are unit tested individually, followed by system testing, in which the tool is brought into the existing OASIS implementation and tested online with other tools. If no problems are identified in testing then the tool is considered integrated into the OASIS implementation.

4. Domain Ontology

The domain ontology constructed for our experiment is shown in Tables 1 to 3. Table 1 informally defines the representational concepts shared by the integration participants. These relational concepts are augmented with tool and service information specific to our OASIS implementation. The *Services Dictionary* (Table 2) outlines the relationship between services, the tools that offer them and the concepts they require. The *Tools Dictionary* (Table 3) indicates the relationship between the tools participating in the integration and the concepts they support. Together these three components succinctly represent all the knowledge related to our OASIS implementation. The term *program object* in the ontology refers to active elements such as procedures, and *data object* refers to data elements such as variables. The concept *Variable* includes all data objects. These distinctions come from the tools involved in the integration. One of the most difficult problems in an integration is the semantic mapping (filtering) between concepts in the shared ontology and those in the integrated tools. In a previous paper [14] we discuss these issues in detail, noting that even subtle differences can yield counterintuitive results.

5. Participant Tools

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

Advanced Software Design Technology (ASDT) Tool. Developed in 1991 as part of a collaboration between Queen's University and the IBM Center for Advanced Studies [5, 16], ASDT provides design recovery and analysis of source code written in *Turing Plus* [10, 11]. ASDT has two main phases. In the *design recovery* phase, Turing Plus code is analyzed to produce a factbase of raw design facts expressed in a proprietary Prolog-like notation. In the *design analysis* phase, the user explores detailed information about entities and their relationships to other entities using the factbase. ASDT provides two tool services that are of interest to our integration efforts:

- *Query*. Given the name of an entity, 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 by transitive closure of 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 explored the use of graph transformations to support maintenance tasks related to software architectures [7]. The Fahmy Tool implements three of these transformations in a tool executed from the command line:

- *High Level Use*. An architecture recovery analysis that promotes low-level *use* relations to higher levels of abstraction in the representation of a software system.
- *Hide Interior*. This service collapses the details of a selected subsystem, hiding its interior components. Relationships among components in the subsystem to/from external entities are preserved.

Table 1. Domain ontology: concepts

System	Represents an entity that organizes or consists of a collection of program objects.
Module	Represents an entity that is a distinct, typically self-contained, program object.
SubProgram	Represents an entity, typically not self-contained, that is stored for use by another program object.
Variable	Represents a data object.
Containment	Represents the relational concept of a program object being contained in another program object.
Use	Represents the relational concept of a program object making use of another program object.

Table 2. Domain ontology: services dictionary

Service	Offered By	Requires Concept
Hide Exterior Hide Interior High Level Use	Fahmy Tool	System Module SubProgram Containment Use
Query Slice	ASDT	System Module SubProgram Variable Containment Use
Spring Layout Sugiyama Layout Visualize	Rigi	*<entity>* *<relationship>*

Table 3. Domain ontology: tools dictionary

Tool	Supports Concept
ASDT	System Module SubProgram Variable Containment Use
Fahmy Tool	System Module SubProgram Variable Containment Use
Rigi	-

Table 4. Characteristics of the OASIS implementation participants

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

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

Rigi Tool. The legacy analysis tool Rigi [1, 18] is the product of more than ten years of research and development at the University of Victoria. While Rigi is a very general tool, our primary reason for including it in our integration experiment 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 layout options that Rigi provides.
- *Sugiyama Layout*. This is a preconfigured 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*. This is a preconfigured 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.

Table 4 summarizes the characteristics of each of the tools chosen to participate in our proof-of-concept experiment. For each tool the following aspects are shown:

- **Programming Language Domain**. Programming language(s) supported by the tool's native parser.
- **Schema Characteristics**. Number of kinds of entities, relationships and constraints in the tool's factbases.
- **Factbase Syntax**. Physical representation of factbases supported by the tool.
- **Services Offered**. Services provided by the tool that are shared in the experimental integration.

Table 4 provides a good indication of the diversity of the tools in our integration experiment. Three different programming language domains are represented: Turing Plus, PLIX and C. The factbase representation of ASDT is detailed and highly constrained, whereas the Fahmy Tool has a much simpler schema with many fewer constraints. ASDT's Prolog-like factbase syntax is completely different from the RSF used by Rigi and the Fahmy Tool.

6. A Step-By-Step Example: Sharing the Fahmy Tool Hide Exterior Service

Software maintenance tasks often focus on a particular component in a complex software system. In this example, we use the *Hide Exterior* service shared by Fahmy Tool to provide support for this kind of analysis on an ASDT factbase. The flow through the OASIS components for each of the steps below is shown in Figure 2. Consider a newly hired software maintainer who has been given the responsibility for updating the parser subsystem of the TXL processor [2], which is written entirely in Turing Plus.

Step ❶ In order to understand the source code better, she begins by using ASDT to perform a design recovery, producing an ASDT factbase.

Step ❷ To take a first look at the factbase, she calls the conceptual service adapter (CSA) for ASDT requesting that it perform the *Visualize* service on the ASDT factbase. The ASDT CSA communicates this request to the Rigi CSA, which creates the domain files required by Rigi and produces the graph shown in Figure 3. Unfortunately, the structure of the parser and even the TXL system itself is lost in the jumble of nodes and edges in this visualization.

Step ❸ In order to focus on the parser subsystem, she decides to use the *Hide Exterior* service provided by the Fahmy Tool. To accomplish this task, she calls the ASDT CSA again, this time requesting that it perform the *Hide Exterior* service on the ASDT factbase.

Step ❹ The ASDT CSA queries the domain ontology, identifies the *Hide Exterior* service and verifies that ASDT supports the concepts that the service requires. The ASDT factbase is mapped to the conceptual space by the ASDT

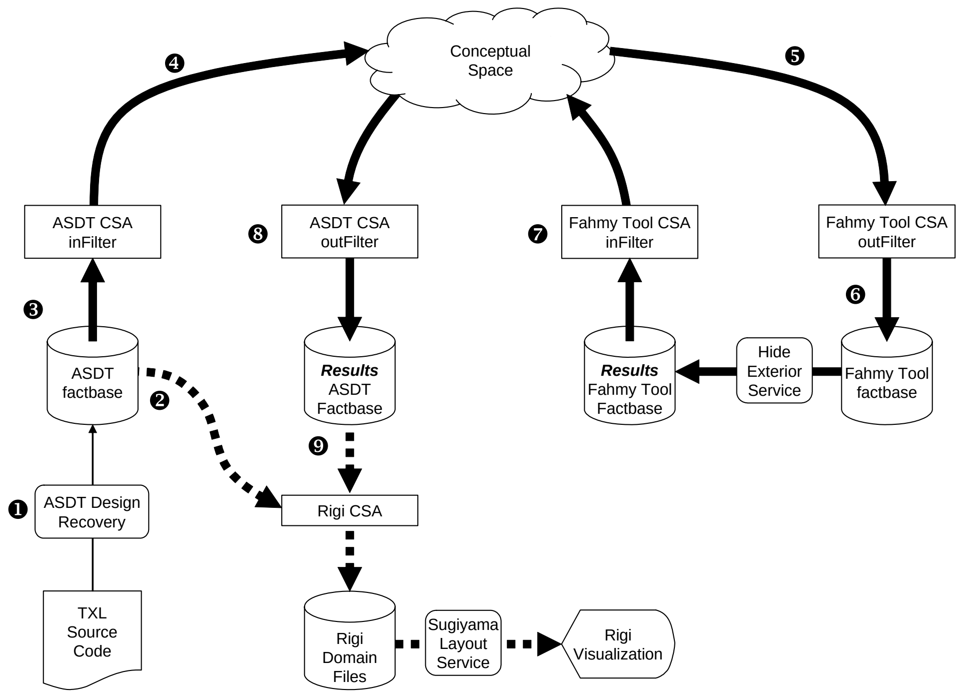


Figure 2. Sharing the Fahmy Tool hide exterior service

CSA *inFilter* and a message is sent to the Fahmy Tool CSA requesting the *Hide Exterior* service.

Step ❺ The Fahmy Tool CSA receives the message and uses its *outFilter* to map the required conceptual space facts to a Fahmy Tool factbase.

Step ❻ The *Hide Exterior* service is invoked by the Fahmy Tool CSA to produce a new Fahmy Tool factbase.

Step ❼ The Fahmy Tool CSA uses its *inFilter* to map the results to the conceptual space and sends a message to the ASDT CSA indicating that the results are ready.

Step ❽ The ASDT CSA receives this message and uses its *outFilter* to map from the conceptual space to an ASDT factbase containing the results.

Step ❾ Hoping for a more reasonable visualization, the TXL software maintainer invokes Rigi's *Sugiyama Layout* service on the results returned from the *Hide Exterior* service. The result is shown in Figure 4. Now a much clearer picture of the structure of the parser can be seen.

This example demonstrates the technique in an analysis of the TXL language processor v6.0 (just over 9,000 Turing Plus source lines, 6,780 ASDT design facts). In other runs we have made use of all of the shared services offered by all three of the tools in our OASIS integration, and we have applied the system to the analysis of larger production software systems such as the Linux Kernel (14,338 Fahmy Tool architecture facts) and the Tobey code generator (over 250,000 PLIX lines, 11,066 Fahmy Tool architecture facts).

7. Related Work

Software analysis tool integration is a difficult problem that has been studied by many groups using many different methods. Most, such as the Software Bookshelf [8], use a central repository and shared schema. The Telos knowledge representation language [19] has been used to implement a common repository at the conceptual level [4], describing a shared global schema in some ways similar to our inferred ontology. GARDEN [21] and FIELD [22] provided integration of tools using the a central message server with 'wrappers' much like our conceptual adapters, and IDL [23] hid representation issues behind a high level interface and a language-independent API for accessing shared data. The problem of differences in factbase format has been widely discussed [6, 20] and attacked by attempts to standardize software exchange notations such as GXL [12].

Our work differs from all these efforts in that we provide a constructive method for building a shared conceptual ontology from the observed concepts used by the tools to be integrated, and a noninvasive method for integrating the tools using conceptual adapters that do not require any reprogramming of the tools themselves. Since integration participants can be off-the-shelf black boxes, our method is well suited to the "COTS" problem [3]. Like IDL, it is independent of schema, technology, environment and system and provides method for integration independent of the nature of the tools. But unlike IDL, it does not require reprogramming to use a shared API.

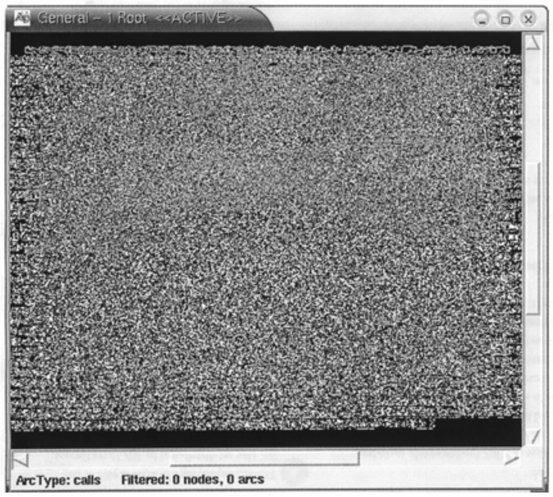

General -- 1 Root <<ACTIVE>>
ArcType: calls   Filtered: 0 nodes, 0 arcs

Figure 3. Visualizing the ASDT factbase

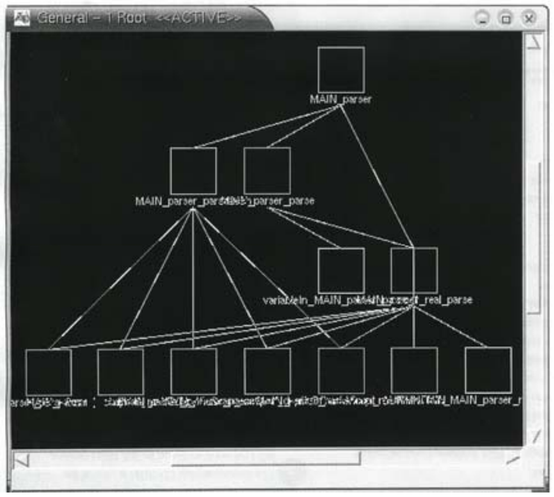


Figure 4. Hide exterior results viewed with Sugiyama layout

8. Conclusion

In this paper we have presented a first experiment in applying the Ontological Adaptive Service-Sharing Integration System (OASIS) integration methodology to the non-invasive integration of services provided by three diverse software reverse engineering tools. Using the integrated system, each tool can invoke shared services on its own native factbase to carry out analyses and visualizations provided by the other tools using different schemas, representations and technologies.

References

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

[2] TXL Home Page. `http://www.txl.ca.`

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

[4] Buss, E. *et al*. "Investigating reverse engineering technologies for the CAS program understanding project." *IBM Systems Journal*, 33(3): 477–500, 1994.

[5] Cordy, J.R. and Schneider, K.A. "Architectural Design Recovery Using Source Transformation". *CASE'95 Workshop on Software Architecture*, July 1995.

[6] Ebert, J., Kullbach, B. and Winter, A. "GraX – An Interchange Format for Reengineering Tools". *Proc. WCRE'99*, pp. 89-98, Feb. 1999.

[7] Fahmy, H.M., Holt, R.C. and Cordy, J.R. "Wins and Losses of Algebraic Transformations of Software Architectures". *Proc. ASE 2001*, pp. 51-60, Nov. 2001.

[8] Finnigan, P. *et al*. "The Software Bookshelf". *IBM Systems Journal*, 36(4): 564–593, Nov. 1997.

[9] Guo, G.Y., Atlee, J.M. and Kazman, R. "A Software Architecture Reconstruction Method". *Proc. WICSA 1999,* pp. 15-33, Feb. 1999.

[10] Holt, R.C. and Cordy, J.R. The Turing Plus Report. Comp. Systems Research Group, University of Toronto, Feb. 1987.

[11] Holt, R.C. and Cordy, J.R. "The Turing Programming Language". *Comm. ACM*, 31(12): 1410-1423, Dec. 1988.

[12] Holt, R.C., Winter, A. and Schürr, A. "GXL: Toward a Standard Exchange Format" *Proc. WCRE'00*, pp. 162-171, Nov. 2000.

[13] Jin, D., Cordy, J.R. and Dean, T.R. "Transparent Reverse Engineering Tool Integration Using a Conceptual Transaction Adapter". *Proc. CSMR 2003*, pp. 399-408, Mar. 2003.

[14] Jin, D. and Cordy, J.R. "Factbase Filtering Issues in an Ontology-based Reverse Engineering Tool Integration System" *Proc. ATEM 2004*, pp. 65-75, Oct. 2004.

[15] Jin, D. and Cordy, J.R. "Ontology-based Software Analysis and Reeng. Tool Integration: The OASIS Service-sharing Methodology" *Proc. ICSM 2005*, pp. 613-616, Sep. 2005.

[16] Lamb, D.A. and Schneider, K.A. "Formalization of Information Hiding Design Methods". *Proc. CASCON'92*, pp. 201–214, Nov. 1992.

[17] Lethbridge, T.C. Requirements and Proposal for a Software Information Exchange Format Standard. Draft manuscript, Nov. 1998. `http://www.site.uottawa.ca/~tcl/papers/sief/standardProposal.html.`

[18] Müller, H.A. and Klashinsky, K. "Rigi - A system for Programming-in-the-Large". *Proc. ICSE'88*, pp. 80–86, 1988.

[19] Mylopoulos, J., Borgida, A., Jarke, M. and Koubarakis, M. "Telos: Representing Knowledge About Information Systems." *ACM Trans. Info. Systems*, 8(4): 325–362, 1990.

[20] Perelgut, S.G. "The Case for a Single Data Exchange Format". *Proc. WCRE'00*, pp. 281-283, Nov. 2000.

[21] Reiss, S.P. "GARDEN Tools: Support for Graphical Programming". *Proc. Int.. Workshop on Advanced Prog. Environments*, pp. 59–72, Trondheim, Norway, 1986.

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

[23] Snodgrass, R., and Shannon, K. "Supporting flexible and efficient tool integration". *Proc. Int. Workshop on Advanced Prog. Environments*, pp. 290-313, Trondheim, Norway, 1986.

[24] Woods, S., O'Brien, L., Lin, T., Gallagher, K., and Quilici,A. "An Architecture for Interoperable Program Understanding Tools". *Proc. IWPC'98*, pp. 54–63, Ischia, Italy, June, 1998.