Re-targetability in Software Tools[*]

Premkumar T. Devanbu,
Department of Computer Science,
University of California,
Davis CA 95616, USA
devanbu@cs.ucdavis.edu
http://castle.cs.ucdavis.edu

September 16, 1999

Abstract

Software tool construction is a risky business, with uncertain rewards. Many tools never get used. This is a truism: software tools, however brilliantly conceived, well-designed, and meticulously constructed, have little impact unless they are actually adopted by real programmers. While there are no sure-fire ways of ensuring that a tool will be used, experience indicates that *retargetability* is an important enabler for wide adoption. In this paper, we elaborate on the need for retargetability in software tools, describe some mechanisms that have proven useful in our experience, and outline our future research in the broader area of inter-operability and retargetability.

1 Introduction

Low productivity and unsatisfactory quality are persistent problems in large software systems projects. CASE tools promise significant improvements in various aspects of software development, including productivity, quality, and repeatability. However, the introduction of new technology in the form of innovative tools, specially in large projects, is fraught with difficulty. While we focus primarily on tool *inter-operability* (the last item listed below) we list some other key failure modes:

Scalability Innovative tools are often not intended for large-scale application; the emphasis, specially in research environments, is often placed on proving the concept in a medium scale system, rather than on production use in large systems. For example, a tool intended for interactive use that runs at about 1-10Klines/minute is not usable in large scale applications with hundreds of thousands or millions of lines of code. Inadequate performance can hinder adoption.

Customizability Software development organizations, specially in large, long-running projects, have localized and specialized development processes that are both designed to meet the needs of the specific application, and adapted to the culture of the organization. Software tools work best if their operation is well-tuned to existing processes. Thus, a tool that generates paper output may not be helpful in an email-based culture.

Familiarity Developers are pressured by schedules, and keenly aware of the need to meet cost, schedule or quality requirements. This engenders a conservative bias towards simple, and/or familiar tools, even if somewhat outdated. Builders of complex tools with steep learning curves (even ones promising significant gains) face the daunting hurdle of convincing busy developers to invest training time.

Inter-operability Large and/or well-established projects have various tool infra-structures in place, such as project code repositories, build procedures, software versioning and defect tracking mechanisms etc. These infra-structures are highly tuned to local needs, and can be very expensive to build and maintain. A software tool that cannot readily inter-operate with such entrenched development infra-structures cannot be adopted easily.

The focus of this paper is on inter-operability and retargetability in software tools; we also argue that as a side-effect of retargetability, sometimes we can make new tools look familiar.

The outline of the paper is as follows. First, we describe the motivations for retargetability. Next, we discuss the difficulties of designing and building retargetable tools. We then present two examples of retargetable tools:

[*]This is a draft, intended for submission to a special issue of the ACM journal *Applied Computing Review*. Please do not circulate without the permission of the author, or the concerned editors.

CHIME and GENOA. Finally, we conclude after a description of some ongoing research.

2 Why Retargetability?

While innovative software tools promise substantial gains in productivity, quality, and interval, there are significant hurdles that have to be overcome before they can be adopted in large software projects. In our research, some of these hurdles have been overcome by making software tools more retargetable. In this section, we describe some issues that are addressed by retargetable tools: *build complexity*, *dialectical variants*, *user-interface familiarity* and *software reuse.*

2.1 Build Complexity

Large, complex software systems can have build procedures that can themselves be very complex. The system may be partioned into literally thousands of header- and source-files. The process of compiling and linking these is controlled by build scripts. These can run into thousands of lines of Makefile-style code. Build scripts can capture different types of dependencies between files, configuration descriptions for different localized versions of the system for different markets, or rules for compiling under different compilers/architectures. Build scripts are typically highly complex and brittle. Relative to their size, maintaining these scripts costs more and is harder than than maintaining code [32].

Consider a new software tool that statically analyzes source code for some purpose (style checkers, test coverage tools etc). Such a tool must be run over all the source files, with the proper include paths, command line options etc set correctly. Adopting such a tool would involve extensive changes the build scripts. The cost of making these changes must be compared with the potential benefits of using the tool itself. In organizations with tight schedules and short investment time-horizons, this represents a major hurdle. A tool that required minimal adaptation to the build procedures has a significant advantage. One way to accomplish this would be build the tool to have a similar "command line signature" to the compiler used in the project; this would require fewer changes to the build process to accommodate the tool. We can do this with a tool that can be retargeted to the existing compiler.

2.2 Language/Dialectic Variance

There are serious efforts underway to standardize popular languages such as C, C++ and Java. However, tool builders are burdened with the significant task of supporting legacy systems that conform to specific dialects of the languages, such as for example the original "K&R" dialect of C, older versions of C++ (2.1), Java etc. The author is also aware of several proprietary variants of "C" that are used in large industrial software projects, which are not compatible with widely available compilers. In addition, consider the fact that there are several popular platforms for C++ development that differ in subtle ways: Visual C++, the GNU C++, the EDG C++ etc. Each of these C++ compilers work best with a particular variant of the required header files; porting files from one dialect to another is a non-trivial effort.

This presents a challenge for tool builders: how does one build a tool that can be used with different dialects of a language? It would be nice if we could simply re-use the compilers and parsers for each dialect. Rather than adapting the tool for each variant, it would be better if the tool could simply attach itself to the existing compiler/parser. This would allow the tool to be run with far fewer modifications to the build scripts.

2.3 User Interface familiarity

Consider a system that is programmed in many different languages. Such a system would involve the use of tools specific to each language: browsers, debuggers, compilers, editors etc. If tools have different user interfaces, the developers are faced with the challenge of mastering several different user interfaces. Clearly it would be best to use the same user interface for each type of tool: *e.g.,* a source code browser which had a similar user interface for browsing all the different languages would be easier to learn and use. Another example: web browsers are ubiquitous and widely known. If we build tools around web interfaces, we can leverage the widespread familiarity with this interface.

A software tool that was retargetable to a familiar user interface is clearly easier to adopt.

2.4 Incompatible repository formats

Many of the different software tools used during the software lifecycle work with the same representation of the relevant artifacts. Thus static analyzers, code generators and interpreters can make use of a common abstract-syntax tree representation. Likewise, cross-referencing, browsing and build-dependency analysis tools can make use of a common symbol table with definition and use information. Using a common data format across different tools saves time and storage space: the format needs to be built only once. For example, the parser need be run only once over the source code to built a persistent AST. It also saves programmer effort in maintaining different versions of the parser for different tools. The common storage manager is also reused by the different tools that need persistent storage.

This fact has not been lost on tool builders. Thus, within the CIA family of tools: cross referencing [7], testing [8] and browsing [7] tools share a common internal repository. The Arcadia environment [17] has a large family of tools (*e.g.,* [26]) that exploit a common internal representation. The REFINE [24] system uses a common shared repository for ASTs. The Rigi [30] family of tools also shares a common repository. There are numerous other examples.

However, representations used by different tool families are not always compatible. Thus the popular and widely used REFINE and Arcadia formats are not compatible; nor are the equally popular CIA and RIGI formats. For the tool builder, this presents a quandary: the use of one of these formats offers many advantages; on the other hand an early commitment to a specific format limits the potential market for the tool. Here again, there is a strong argument for building tools that are retargetable to different shared repository formats.

2.5 Software Reuse

Lurking in all the discussions above is an underlying concern with software reuse. Retargetable software tools can usually *reuse* a large software infra structure: repositories, parse trees, build scripts etc all embody valuable investments in software, which the tool builder or tool adopter will have otherwise to re-create.

All the well-known advantages of software reuse [19] accrue to the retargetable software tool. Risks, costs and interval are reduced by avoiding development of critical elements. Quality is also enhanced (in many cases) my re-using tested, mature code. If the reused, existing software has external user-interfaces, the users' familiarity with the existing software is leveraged.

3 Retargeting Barriers

There are many advantages to designing software tools to be retargetable. However, there are difficult design and implementation issues that must be faced while constructing a retargetable tool. We describe some of these below, using the example of a static analysis tool.

Consider implementing a static analysis tool to perform coding conventions checking on a source program [12]. Such a tool typically starts with the abstract syntax tree (AST) of a program. Many compilers build a fairly complete AST which they use ultimately for code generation. Hence, for all the reasons mentioned above, it would be desirable to build source code analysis tool around such a compiler; indeed, it would be desirable to build a source code analysis tool to be retargetable, so it can work with an existing compiler. However, this is not always easy to do. We describe several types of difficulties:

3.1 Closed Legacy Software

In a heterogeneous, component-based, networked world, people building and using software systems see "openness" as an essential property. An open system can communicate with other systems and mutually enhance functionality. Systems are built to be accessible via one or more standards. ODBC (Open Database Connect), HTTP, CORBA, COM, XML are all popular standards to which open systems adhere.

But it was not always so. Before networking and component-based development, systems were built by single vendors to run on a single machine for a single purpose; market incentives such as the desire to "bond" with the customer actually indicated that systems could *not* be open. Examples abound:

1. Telephone switches that write billing records in proprietary formats to tapes that have to be hand-carried to proprietary billing systems;
2. Clinical information systems [28] which do not inter-operate with hospital billing systems;
3. Complete chemical process-control systems incorporating both measurement and control, but which only communicate externally through tape logs and consoles;
4. Compilers and source code analysis tools, both of which build and use abstract syntax trees, but do not inter-operate.

Systems that are closed are not easy to inter-operate with; thus, it not easy attach them to retargetable tools.

3.2 Interface Design

"Attaching to" a closed legacy system can be conceived as opening a window to its inner workings, so that another system may receive information from it, as well as affect its state. It may be desirable to provide an inter-operable interface to the system, perhaps in the context of a standard such as CORBA [22] or COM [1]. This can be a non-trivial design problem. Interface design must balance several conflicting goals:

Simplicity The interface must be simple, well-documented, and comprehensible to both the clients of the interface and the implementors. An overly complex interface with poor or inconsistent documentation is unlikely to be useful.

Functionality A large closed legacy system typically provides a large and diverse set of functions. It may be desirable to access many of these functions via with open standards. Unfortunately, more functionality leads to large, complicated interfaces. For a complicated closed legacy system, the trade-off between simplicity and functionality complicates interface design.

Extensibility Inevitably, a successful effort at "opening" up a closed legacy system leads to increased use. This usually leads to calls for changes and extensions to the original interface; initial interface design should consider the likely need for later extensibility.

Compatibility/Implementability The interface design must be implementable; there should be any fundamental reason why such an interface is difficult to build. This falls under the issue of architectural mismatch, which we consider next.

3.3 Architectural Mismatches

The study of software architecture, beginning with [23] has emphasized the importance of styles [27]. In particular, Garlan [13] has explored difficulties arising out of *architectural mismatch.*. Two systems based on incompatible architectural styles can be difficult to bring together. For example, if two C or C++-based systems, both of which include a "`main`" routine have to be integrated into one executable, some re-structuring of the code will be required. As another example, a system based on an interrupt-driven style may be difficult to integrate together with a system programmed in an event-loop based style.

A more complex example: consider a process-control system, which monitors and controls a chemical plant. The system uses a standard real-time system architecture with a single process handling a fixed number of tasks at differing priorities, and with different deadlines. High levels of reliability would be required, and the (legacy) software might have been developed with safety-criticality in mind. It is desired to make this system web-accessible; a web client should be able to view the status of the system. To do this, a web server needs to be integrated with the system. This presents various difficulties. The existing system design does not allow multiple processes; so a web server cannot be simply added to the system. So it would be necessary to hand-code the web service as an additional task within the existing roster of tasks handled by the lone executing process. Issues such as the priority of the web service task, the handling of time-outs (in case the web service is pre-empted by other tasks) etc. would have to be carefully considered.

In summary, while retargetability offers several advantages, there are some serious difficulties that arise in any specific effort.

4 Retargeting Experiences

We have and continue to build retargetable software tools. In this section, we describe our experiences with different systems, and the lessons learned.

4.1 The CHIME Experience

Program understanding is a significant and time-consuming component of software development. Source code browsers support this task by exposing implicit relationships in the source code (between function call sites and the definitions of the functions, for example) as hypertext links. Software development environments often provide built-in hypertext browsing for source code. However, few of these source code hypertext systems are as powerful, sophisticated, compatible, dynamic and feature-rich as the the world-wide web. The goal of the Chime [11] system is to bring the rich (and getting richer) infra-structure of the world-wide web to existing software development environments. Chime is a domain-specific framework that generates *link-insertion* engines that insert HTML links into source code. Links can, for example, connect a function call in the source code to the function definition. A complex language like C++ admits many such relationships. Chime includes a link specification language in which links can be specified. The user can control both the *position* and the *semantics* (i.e., what should happen when the link is activated?) of a link. Different program understanding tasks require different reading tactics [20]. Chime allows the creation of different "views" of a source file, that expose different links; thus, it can be tuned for different tasks and different users. Chime needs certain information to insert these links: for example, the position of function calls, and the location of function definitions. This information is typically stored in repositories in software development environments. As discussed above in Section 2.4, different repositories use different formats; this makes it difficult to get the needed information out. Chime addresses this problem by adopting an interface very similar to the open database connect (ODBC) standard for accessing databases. This interface includes facilities for querying databases, iterating over the tuples in a database relation, and accessing attributes in a tuple. To use Chime in conjunction with an existing software repository, it is sufficient to implement this interface. Once this is done, it is possible to specify several different types of views comprising different groupings of HTML links; users can then browse source code with WWW clients using these views.

The core retargetable component of Chime is the repository interface. As discussed in Section 3.2. This presented several challenges. There are several possible variations in repositories. Different data models could be used: relational, entity-relationship, object-oriented, semantic, hierarchical, etc. Each of these models would require a different style of interface for data access: *eg.,* the notion of "relation" and "tuple" does really exist in a semantic data model. We felt that an interface to accommodate all these different data models would make the interface, as well as the Chime system very complex. So we chose to accommodate a relational style, with some extensions for set-valued attributes. While different relational databases provide somewhat different APIs, we settled on an ODBC-style interface, which is mature and has broad coverage. Clearly, there is a trade-off here: it would be easiest to implement this interface for relational repositories; others would be not as easy. On the positive side, with reference to the discussion in Section 2, the retargetability aspects of Chime provide several advantages.

1. **User Interface**: WWW browsers are almost universally familiar interfaces This familiarity can be fruitfully transferred over to source code browsing.

2. **Repository Format:** Chime takes a simple, uniform view of repositories, via an interface. This simplifies the problem of adapting to new repositories (Section 2.4): it is only necessary to implement this interface for each new repository format. Chime has been retargeted to several formats, including both the CIAO [7] and the Rigi [30] formats.

3. **Software Reuse**: In addition to the formidable, sophisticated, rapidly evolving base of software available for the WWW, the software used to construct the repository is leveraged.

4. **Build Complexity**: By reusing the existing repository, Chime precludes the need to modify the build procedure in order to introduce a new tool. The use of the existing repository also avoids the need to construct parsers for different dialects of programming languages etc.

4.2 The GENOA/GENII Experience

Consider the problem of building static source code analyzers for C++ programs. Here are some typical tasks:

1. For each file, print all locations where variables are modified, and where they are simply accessed (without regard to aliasing, pointers etc).

2. Find method calls of all types (constructor, destructor, overloaded operator, etc) and report their locations (file, line number).

3. Report all cases where a destructor is not virtual.

The tasks above involve reading in the source code, performing lexical analysis, parsing, scoping analysis, type checking etc; finally, the resulting annotated AST can be processed for the indicated analysis task. While a tool specialized for each of the tasks above would be hard to build, there is much in common across all such tools. In fact, these tools would only differ by the final processing done with the decorated AST. In GENOA, the above examples (and other similar tools) are implemented by writing an AST traversal in a high-level language that is specialized for processing ASTs. One may think of this language as "a query language for ASTs". Most of the above tools can be implemented in a few lines in this language. Full details of GENOA can be found in [10], and the software is available free [9].

Conceptually, GENOA rests on the notion of querying ASTs, and is independent of a specific language, or a particular representation of an AST in terms of specific data structures. Given a specific language on a specific platform, it is necessary to tokenize and parse the language to construct the abstract syntax tree. In some cases, as with over-loaded operators in C++, it is even necessary to use semantic information in order to recognize something as an over-loaded operator. Languages like C++ have complex interactions between syntax and semantics. For example, the declaration "`typedef int FOO`" in C and C++ changes the lexical role of the token "`FOO`" from an identifier to a type name. Constructing a parser to build an AST for C++ is quite a time-consuming task, requiring a great deal of manual effort beyond the support provided by tools based on purely grammatical descriptions such as Yacc [16], CENTAUR [3], Refine [24], etc. In addition, there are many dialectical variants of C and C++, with their own idiosyncrasies. Thus there are many good reasons to attempt to reuse an existing AST builder. GENOA relies on a subsystem called GENII, which makes the AST querying mechanism retargetable.

```
1 Funcall:(IS-A Expression)
2    ’p->nodetype == E_FUNCALL’ ‘‘UNARYEXPR *’’ {
3    atline:  an Integer  <’’p - >getLine ()’’>
4    atFile:  a String    < ’’p - >getFile ()’’>
5    callname:  a String  < ’’p - >funName ()’’>
6    args:  an OrderedContainer of Expression
                             <’’p->getArgs()’’>
7 }
```

Figure 1: Sample GENII specifications

The GENII language is designed to model AST implementations. Some sample GENII specifications for a function call AST node are shown in figure 4.2: This specification indicates that a function call (line 1) is a kind of `Expression` (defined elsewhere, not shown). In this implementation it is represented (end of line 1) by a data structure (line 2) of type `UNARYEXPR *`. Assuming that a variable `p` is of this type, the test `p -> nodetype == E_FUNCALL` can be used to check if `p` really points to a `Funcall` node. A function call node includes information about the line number and file name where it occurs (lines 3,4) the name of the function being called (line 5) and a list of arguments (line 6). The actual code to extract this information is shown within "`< ... >`". Of course, a full GENII specification describing all the details of an AST implementation is quite large. For example, the GENII specification that retargets GENOA to a C++ compiler includes descriptions of about 300 AST node types, and is about 1600 lines long. This approach has been used successfully to retarget GENOA to 3 different C++ compilers, one C compiler, and one Java parser. We have realized several advantages with this approach, as summarized in Section 2

1. **Build Complexity:** As described in Section 2.1, the use of an existing parser/compiler in GENOA analysis tools is a major advantage. In the case of GEN++ [9], which is based on a widely used C++ compiler, generated analysis tools have exactly the same "command-line signature" as the compiler. Build procedures can invoke the source code analysis tools exactly as they invoke the compiler to compile the source code. In our experience, incorporating a GEN++ tool into an existing compiler typically involves changing only a few lines in the make scripts.

2. **Software Reuse:** Most software analysis tools use a custom C++ parser. Tools such as CIA [6] and Cscope [29] use this approach. As discussed earlier, this can be a difficult and time-consuming task. It is better to re-use and adapt an existing parser: e.g., CIA++ [14] uses the parser component of a C++ compiler. However this adaptation process must be carried out for each analysis tool. With the GENOA approach, we model the data structures in an existing parser using the GENII language, and use this model to construct an entire range of analysis tools.

3. **Language/Dialectic Variance:** GENOA offers a solution to the problem (Section 2.2) of dialectic variance: by retargeting the parse tree querying engine to a parser for a specific dialect, it becomes possible to produce an analysis capability for that dialect. In general, is better to re-use an existing, validated, well-proven parser for a dialect rather than developing a new one (or adapting an existing one to suit the new dialect).

These advantages of retargetability were obtained as a result of careful design trade-offs that were made in order to address the difficulties in constructing retargetable tools.

1. **Closed Legacy Systems:** GENOA can make use of ASTs constructed by systems where the AST not intended to be reused in other contexts. For example, one port of GENOA, *viz.,* GEN++, makes use of a widely used C++ *compiler*, that has a very complex AST representation, consisting of several dozen classes and a few hundred different methods. The compiler was intended to be a closed system, not designed for interoperability with other systems. The AST representation was designed specifically for use within the compiler, and thus simplicity and interoperability was not a goal. The GENII system can be used to model this complex representation once and for all; after this, the complexity can be hidden from the tool builder. In this manner, the intricate details of the closed legacy compiler are made open for use by a whole family of source code analysis tools.

2. **Interface Complexity:** GENOA takes a very specific view of ASTs that is suitable for source code analysis tools. Analysis tools need to traverse an AST node and its descendants, unparse and print the node, identify the corresponding location in the source code, etc. The GENII interface description must identify code within the legacy parser that performs all these functions. As shown in figure 4.2, there are functions provide to traverse the different components of an AST node (the called function name associated with a function call, etc). GENII also provides a simplified model of collections and lists, such as the statements in a function body or the arguments to a function call. However, GENOA is designed mainly for analysis tasks; so the interface that GENII provides to the legacy AST is quite restricted. For example, GENOA does not allow tool builders to modify the AST. In addition, the access to the AST is navigationally restricted. There is no notion of propagating values up or down the AST, as can be done with attribute-grammar based approaches [25]. Thus, if an analysis tool is exploring one portion of the AST, there is no systematic way to look at a related part of the AST; in general, such non-local accesses must be done using standard stack or global variables. However, in practice, the compromises made in this interface have not proven a barrier to users; a range of applications have been reported [2, 4, 5, 31, 21, 15, 18]

5 Future Work: retargetable debuggers

Our current research is focused on a different task: debugging programs. Specifically, we are interested in debugging domain specific languages. We briefly describe this research before concluding the paper.

The software industry is under great pressure to reduce costs, increase quality and shorten development intervals while simultaneously creating products which are more customizable. Industry has found it difficult to meet these conflicting goals by building systems with conventional programming languages. Domain Specific Languages (DSL)'s, which are high-level languages with constructs tuned to express concepts in a specific application domain, have emerged as a viable approach. Entire applications can be generated from short scripts in a DSL. DSLs can be implemented either by compilation to a language like C or Java, or by interpretation. But once a DSL is implemented, developers using that DSL face a downstream problem: debugging applications written in DSLs. This problem has not received much attention.

DSL applications can be large. GENOA, for example, has been used for complex tasks such as control dependency analysis and path condition generation. Such GENOA programs can be hundreds of lines long, and include complex control flow and data dependencies. In such cases, defects inevitably creep in. Defects cause various failures: bad output, infinite loops, or run-time errors such as popping an empty stack. When failures arise, the DSL user may try to isolate the problem by reading the DSL program and mentally "simulating" its execution. This is hard to do for large programs. The user may also try inserting "print" statements to display intermediate state; but this requires repeated re-compilations, and risks leaving stowaway debugging "print"s in the final product! Interactive debugging is a powerful way to isolate defects in programs. However, debuggers are expensive and difficult to build, and it is typically not economically feasible to construct debuggers for each domain-specific language. So DSL users are typically left without interactive debugging facilities. The question arises, can we construct a *reusable framework* that can be leveraged to provide debugging support for different DSLs?

We approach this situation by first observing that all debuggers have several functions in common—first, a user interface (graphical or otherwise) allowing users to set break points at specific points in the code, and also to inspect the state of the application being debugged. They may also have an event pattern recognizer, which allows a specific pattern of events to observed. We also note that several DSLs are implemented via interpreters. Our approach then, is to construct a *retargetable* debugging framework that can be attached to an existing interpreter for a DSL. This framework provides such common functionalities as a user interface, event pattern recognizer etc. This framework is attached to a specific interpreter much in the same way as GENOA is attached to a specific parser or AST builder. There is a retargeting subsystem, where one models the data structures used by a specific interpreter to represent the state of the running program. This model is constructed in a specification language similar to GENII. From this model, the retargeting machinery is generated. The retargeting machinery then provides the debugging framework with a uniform, simple way to access the running program's state. Additional modifications to the interpreter will be required to enable the debugger to start and stop the interpreter at specific points in the interpreted program.

The advantages here are typical of a retargetable tool: First, there is a significant amount of code reuse. Both the interpreter and the debugging framework are leveraged. Second, the interpreter's entire run-time is available not only to inspect the running programs state, but potentially also to evaluate expressions, etc.

Several potential hurdles also exist. First, to simplify the retargetable tool framework, it is necessary to design a uniform, simple interface to the interpreter's runtime datastructures. This will bring with it some limitations on what aspects of the state can be inspected. Second, DSL interpreters are likely to be closed legacy systems, without any consideration given to allowing reuse in a different context. There may be architectural reasons why some interpreters will not work with a given design for the debugging framework. This work is ongoing.

6 Conclusion

In this paper, we have identified some key obstacles to widespread adoption of tools. We focus on retargetability and inter-operability. We explore some of the advantages of retargetable tools, and describe some of the obstacles to retargetability. We present our experience with two retargetable tools, Chime and GENOA/GENII. Finally, we describe our current research with retargetable debuggers for DSLs.

There has been great innovation in software tools for verification, testing, metrics, coding standards etc. Retargetability is an important feature that will support more widespread exploitation of these innovations.

References

[1] *ActiveX Consortium.* `http:/www.activex.org`.

[2] J. Bieman and B-K. Kang. Cohesion and reuse in an object oriented system. In *Proceedings Proc. Symposium on Software Reusability (SSR'95)*, 1995.

[3] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pasual. Centaur: The system. In *Proceedings of the Symposium on Software Development Environments*, 1988.

[4] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for c++. In *Proceedings, Nineteenth International Conference on Software Engineering.* IEEE Press, 1997.

[5] L. Briand, S. Morasca, and V. Basili. Goal-driven definition on product metrics based on properties. Technical Report CS-TR-3346, University of Maryland, Computer Science Department, 1995.

[6] Y. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy. The c information abstraction system. *IEEE Transactions on Software Engineering*, 16(3), March 1990.

[7] Yih-Farn Chen, Glenn S. Fowler, Eleftherios Koutsofios, and Ryan S. Wallach. Ciao: A Graphical Navigator for Software and Document Repositories. In *International Conference on Software Maintenance*, 1995.

[8] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. Testtube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, 1994.

[9] P. Devanbu. The GEN++ page. `http://seclab.cs.ucdavis.edu/~devanbu/genp`, 1998.

[10] P. Devanbu. Genoa - a customizable, front-end retargetable source code analysis framework. *ACM Transactions on Software Engineering and Methodology*, 9(2), April 1999.

[11] P. Devanbu, R. Chen, E. Gansner, H. Muller, and A. Martin. Chime: Customizable hyperlink insertion and maintenance engine for software engineering environments. In *International Conference on Software Engineering*, 1999.

[12] C. K. Duby, S. Myers, and S. Reiss. Ccel: A metalanguage for c++. Technical Report CS-92-51, Dept. of Computer Science, Brown Univeristy, 1992.

[13] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch, or, why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering.* IEEE Computer Society, May 1995.

[14] Judith Grass and Y. F. Chen. The C++ Information Abstractor. In *The Second USENIX C++ Conference*, April 1990.

[15] D. Jerding, J. Stasko, and T. Ball. Visualizing message patterns in object-oriented program. In *Proceedings, Nineteenth International Conference on Software Engineering.* IEEE Press, 1997.

[16] S.C Johnson. Yacc — yet another compiler-compiler. Technical Report 32, Bell Laboratories, 600, Mountain Ave., Murray Hill, NJ 07974, July 1975.

[17] R. Kadia. Issues Encountered in Building a Flexible Software Development environment: Lessons from the arcadia project. In *Proceedings of the SIGSOFT Symposium on Software Development Environments*, 1992.

[18] S. Karstu and L. Ott. An investigation of the behaviour of slice based cohesion measures. Technical Report CS-TR 94-03, Michigan Technical University, 1994.

[19] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 28(2), 1996.

[20] S. Letovsky. Cognitive processes in program comprehension. In *Proceedings of the Second Workshop on Empirical Studies of Programmers*, Washington, DC, 1986. Ablex Publishers, Norwood, NJ.

[21] N. C. Mendonça and J. Kramer, editors. *Proceedings of the Workshop on Program Comprehension*, Los Alamitos, California, April 1998. IEEE Computer Society, IEEE Press. 1996.

[22] OMG. The common object request broker architecture (CORBA) `http://www.omg.org/`, 1995.

[23] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, October 1992.

[24] REASONING SYSTEMS, INC of Palo Alto CA. *REFINE User's Guide.* 1989.

[25] T. Reps and T. Teitelbaum. The synthesizer generator. In *Proceedings of the Symposium on Software Development Environments*, 1984.

[26] D. J. Richardson, T. O. O'Malley, C. Tittle Moore, and S. Leif Aha. Developing and integrating prodag in the arcadia environment. In *Proceedings of the SIGSOFT Symposium on Software Development Environments*, 1992.

[27] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[28] Warner Slack. *Cybermedicine: How Computing Empowers Doctors and Patients for better Healthcare.* Jossey-Bass, 1997.

[29] J. Steffen. *The CScope Program, Berkeley UNIX Release 3.2*, 1981.

[30] Margaret-Anne Storey, Kenny Wong, and Hausi A. Mueller. Rigi: A visualization environment for reverse engineering. In *Proceedings of the 1997 international conference on Software engineering*, 1997.

[31] S. Woods and A. Quilici. Some experiments toward understanding how program plan recognition algorithms scale. In *Proceedings of the Working Conference on Reverse Engineering*, Monterey, CA, October 1996.

[32] S. Zeigler. Comparing development costs of C and ADA. `http://sw-eng-falls-church.va.us/AdaIC/docs/reports/cada/cada_art.html`.