GENOA - A Customizable, front-end retargetable Source Code Analysis Framework

Premkumar T. Devanbu
devanbu@cs.ucdavis.edu

Dept. Of Computer Science, University of California, Davis, CA 95616

Code Analysis tools provide support for such software engineering tasks as program understanding, software metrics, testing and re-engineering. In this paper we describe GENOA, the framework underlying application generators such as Aria [Devanbu et al. 1996] and GEN++ [Devanbu 1998] which have been used to generate a wide range of practical code analysis tools. This experience illustrates *front-end retargetability* of GENOA; we describe the features of the GENOA framework that allow it to be used with different front ends. While permitting arbitrary parse tree computations, the GENOA specification language has special, compact iteration operators that are tuned for expressing simple, polynomial time analysis programs; in fact, there is a useful sublanguage of the GENOA language that can express precisely all (and only) *polynomial time (PTIME)* analysis programs on parse-trees. Thus, we argue that the GENOA language is a simple and convenient vehicle for implementing a range of analysis tools. We also argue that the "front-end reuse" approach of GENOA offers an important advantage for tools aimed at large software projects: the reuse of complex, expensive build procedures to run generated tools over large source bases. In this paper, we describe the GENOA framework and our experiences with it.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.6 [**Software Engineering**]: Programming Environments

General Terms: Languages

Additional Key Words and Phrases: source analysis, reverse engineering, metrics, code inspection

1. INTRODUCTION

A big part of the cost of maintaining large systems is the effort spent by programmers to comprehend unfamiliar pieces of code. Corbi [Corbi 1989], based on practical experience at IBM, reports that programmers can spend 50% of their time on program comprehension. This comprehension task has been called *discovery* [De-

Address: Room 2063, Engineering Unit II, 1 Shields Way, Davis, CA 95616
This a revised and extended version of an article that appears in the proceedings of *12th International Conference on Software Engineering, 1992*

vanbu et al.]. In a previous paper on the Lassie [Devanbu et al.], we argued that knowledge about a large software system can be captured in a *software information system* (SIS), and made available to assist programmers. One of the most important issues raised in our work is the difficulty of acquiring the knowledge for a SIS. *Fixed analysis tools* such as CIA, CSCOPE [Chen and Ramamoorthy 1986; Steffen 1981], etc., extract information directly from the source code of the system. These tools perform a partial, targeted scan of the code, and produce predefined reports (perhaps directly into a database). Although these tools are useful, the information they extract is fixed; as we shall see below, there is often a need for information that is syntactically extractable, but is not available from existing analysis tools. Much useful information sought by programmers during discovery can be obtained simply by static analysis of the source code. Here are some examples (pertaining to C++ code):

(1) Are there any switch statements in this code that don't have a `default:` case ?
(2) Do all routines which switch on a variable of enumeration type `TrunkType` handle the `ISDN` case?
(3) Do any routines which call the `SendMsg` routine, default on the last 3 arguments?
(4) Which routines call *only* the routine `SendMsg` and don't have references to data of type `TrunkData`?
(5) Does any subroutine redeclare a variable in a contained context with the same name as a parameter or a global variable?
(6) Which functions are declared both `virtual` and `inline`?
(7) Is there a call to the function `SendMsg` routine that is control dependent on a call to the function `isProcessAlive`?

Questions such as these can be answered by analyzing the syntactical structure of the source program[1]. In most cases, this would involve parsing the source code[2] to yield an abstract syntax tree, and then traversing the abstract syntax tree to generate the needed information. The only difference between the different questions is the nature of the abstract syntax tree traversal required to answer them.

In addition to program understanding, there are a wide range of applications for static analysis in metrics, testing, inspection, etc. The goal of the GENOA framework is to simplify the implementation of static analysis tools. We would like to specify particular kinds of analyses that would be performed on an abstract syntax tree, with a simple *scripting language*. From this specification, an executable analyzer would be generated. This analyzer could be run over source files. This scheme is illustrated in Figure 1. Source code is lexically analyzed, parsed, perhaps type-checked and semantically analyzed by a *front-end*, which builds an abstract syntax tree; a specification of how to traverse this is translated into a *back-end*. Together,

[1] Some of the examples shown above can involve pointer aliasing analysis, e.g., function calls can be indirected through a call table. In such cases, we assume a conservative approximation can be accomplished by syntactic analysis: for example, all functions with the same signature could be reported as possible aliases.
[2] Some tools such as LSME [Murphy and Notkin 1996], trade-off accuracy for efficiency and "tolerance" by ignoring parsing altogether. We discuss this later.

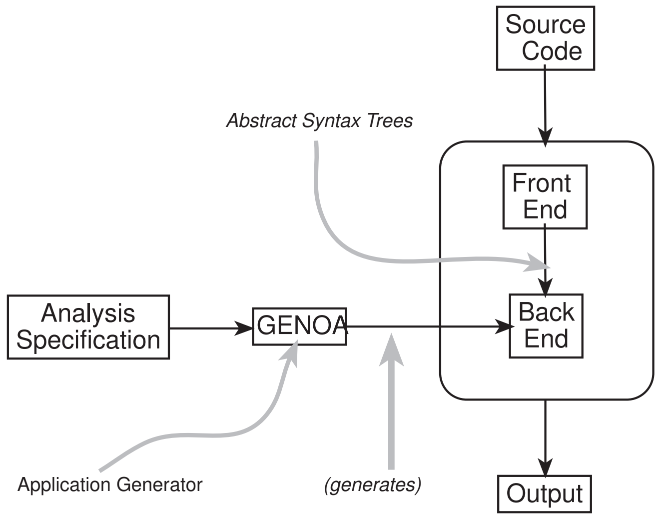

Fig. 1. A Source Analyzer Generator

the front- and back-end constitute the analyzer (inside the large box with rounded corners).

There are important considerations in the design of this scripting language. It should be simple and concise (analysis task scripts should be compact); it should also have concise operators that are designed to express iterated traversals of abstract syntax trees (e.g., *"Find all the assignment-statements"*) and to move to the some specified child of a node in the abstract syntax tree (e.g., *"go to the left hand side of an assignment"*). Its computational properties are also important; analyzers generated from queries could be run over many hundreds of thousands of lines of code, so it would be desirable for the scripting language to have an efficiently computable subset. This subset should also be powerful enough to express many of the questions that programmers might want to ask about large bodies of code. The language should have transparent semantics so that, given a script, it is easy to explain what it does, and to make a good guess about its execution time. If it is hard to predict the running time of a given script, the toolsmith or a programmer would be left uncertain about how long a given query might take to run over a large body of code. The GENOA language attempts to address these issues: it's a simple, compact language, with straightforward procedural semantics. To see an example of an analysis specification in the GENOA language, see Example 1 beginning on page 10. We also show in §7 and §8 that there is a subset of the GENOA language which can express all (only the) polynomial time computations (on the size of the parse tree). The expressiveness result in §8 shows that this subset captures a wide set of useful queries (a great many useful tools turn out to be no worse than polynomial in the size of the parse tree). Furthermore, these results give a good intuition about how to estimate the complexity of queries written in the subset of the GENOA language.

In addition, there are two important pragmatic details. First, toolsmiths building source analyzers for popular languages, like C and C++ have to overcome a major hurdle: these languages have many irregularities in their lexical, syntactic and type structures; this leads to a lot of intricate special-case handling during parsing. Toolsmiths have to confront the unpleasant task of either constructing a parser or adapting an existing parser. Second, it can be difficult to incorporate new tools into an existing large project. There may be thousands of source files, header files, and configuration parameters. Build procedures which run compilers over a large source base can be very complex and brittle. There are intricate makefiles and build scripts which often run into thousands of lines, and require several full time support staff. The build procedures are often more expensive to create and maintain than the source code itself, and are particularly complex for C and C-like languages (See, for example [Zeigler]). These build procedures depend on the specific compiler that is being used: compilers have specific *environmental factors* such as command-line options, include files, and "-D" options. Converting these build scripts to run a tool (which makes different environmental assumptions) over thousands of files, with complex compile time options and numerous include files, is a major undertaking. The cost of this modification can often exceed the benefits of using the tool. Tools based on GEN++ (which uses the the Cfront parser) accept the same command line options as the standard Cfront-based compiler.

By re-using an existing parser, we save the effort of re-building a parser, *and* produce tools which mimic the environmental behavior of the parser. These tools are easier to integrate into existing environments. Here is an executive summary of the GENOA framework's contributions:

—It is designed to be language-independent and parser-retargetable; it can be interfaced to a wide range of parsers for different languages (the limits of retargetability are discussed in §6). In this way, we both avoid reconstructing the parser, and build on the quality, features and coverage of an existing, validated system. The interface (to the existing parser) itself is generated from a formal interface specification, using a companion system called GENII, which is described in §6.

—An important benefit of re-using an existing parser is that the parser's "command-line signature" is reflected in tools, and thus existing build procedures can be re-used to run tools over large, complex source bases.

—GENOA provides a compact scripting language which facilitates the implementation of tasks such as the ones listed above.

—This language is simple, and has transparent semantics. We demonstrate this empirically with several examples, and analytically by showing the complexity and expressivity of a useful sublanguage.

In this paper, we first motivate both the "parser-retargetable" architecture of the GENOA framework by describing the difficulties inherent in constructing a source code analysis tool, and the simple scripting style used in the GENOA language. We then introduce the GENOA language with a detailed example which illustrates the analysis of C programs. Afterwards, we describe how GENII can be used to

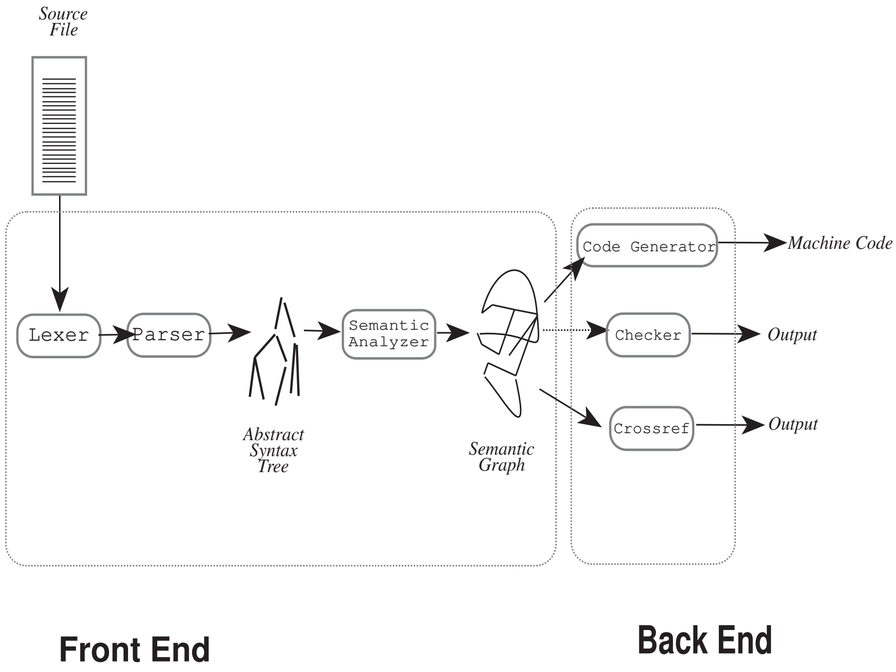

Fig. 2. The internal structure of code analyzers

instantiate the GENOA framework to other languages/front-ends. We then conduct a more detailed exploration of the GENOA specification language, and identify a useful subset that has polynomial time complexity.

2. RELATED WORK: SOURCE CODE ANALYZERS

Consider the conventional design for the general category of tools that process source code. These range in function from simple cross-referencing tools, such as Cscope [Steffen 1981] to complex tools, such as the ones that construct program dependency graphs or style checkers such as Lint. All of these tools have a very similar internal structure (See Figure 2). There is a "front end" including a lexical analyzer, parser, and semantic processor. The lexical analyzer reads the input file and turns into a stream of *tokens*, which are processed according to the grammar of the input language, and arranged into a parse tree (and/or abstracted into an *abstract syntax tree*, or AST). The AST is processed to extract semantic information such as the scope, type and value of symbols; this information is captured and entered as annotations into the AST, resulting in an *abstract semantics graph* (ASG) (See [Rosenblum and Wolf 1991] for a description of this term). Symbol references may point "back up" the AST; in general, we have a graph.

After this point, the "back end" performs a specialized traversal of the ASG, extracting information that is relevant to the particular task. We show three examples in the right hand side of Figure 2, including a code generator (as in a compiler), a checker (as in Lint) and a cross referencing tool (such as CIA).

Systems such as CIA and Cscope are not customizable, and have a fixed task to perform; so they don't need a full syntactic and semantic analysis of the code or a fully attributed ASG. Their "front end" is tuned to the particular task and builds only an abridged representation of the source code suited just for that specific task. Systems such as Rigi [Storey et al. 1997] are more flexible, and can be customized to extract different program representations. However, to extract new information from the source code, it would necessary to modify the parser. In order to answer arbitrary questions about the structure of the code, such as the ones posed in § 1, it is necessary to make a *full* parse of the code, and construct a *full* ASG; an analyzer can then walk over the ASG and extract the information desired.

There are two subsystems needed here: the front end, that builds the ASG and the back end, that traverses this tree in a specifiable manner. There are several existing systems that are potentially relevant to each of these tasks, but (as we shall see) there are complications. A key difficulty in building front ends is that popular languages, such as C and C++, are syntactically and semantically irregular. Certain features of C and C++ conflate the different syntactic and semantic aspects of front-end processing. There are several tools which provide suitable abstractions (such as context-free grammars) that help specify and automatically generate the different phases of front-end processing (such as lexing, parsing, static type inference, etc). However, the syntactic and semantic intricacies of C and C++ demand a great deal of special-case handling which makes it difficult to cleanly separate the different phases, and create a simple implementation.

The $UNIX^{TM}$ tools Lex and Yacc are useful for building lexers and parsers, respectively. Yacc can be used to build context-free grammar (CFG) parsers. Many popular programming languages (such as C) aren't context free, so Yacc provides "semantic actions" that can be used to access symbol tables, etc. More modern tools, such as the Pan system [Ballance et al. 1990], Metatool [Cleaveland and Kintala 1988], Eli [Gray et al. 1992] CENTAUR [Borras et al. 1988], Gandalf [Habermann and Notkin 1986], and the Cornell Synthesizer Generator [Reps and Teitelbaum 1984] provide an integrated environment to implement syntax/semantic processing. In addition to a parser-generator, they provide ways of implementing semantic processing. The CENTAUR system has two methods, one based on a tree manipulation language (VTP) and the other based on a natural-deduction style semantic specification (TYPOL). Other systems use attribute grammars [Reps and Teitelbaum 1984; Gray et al. 1992]. GANDALF uses a programming language (with tree manipulation data-type) called ARL; Pan can perform similar functions. The tools mentioned above could theoretically be useful in building source analyzer generators; but (for all the reasons mentioned in the previous paragraph) they are not as helpful with languages such as C and C++. In order to use these tools, it would be necessary to implement within their special context a fully functioning front-end; but it would be preferable to simply reuse an existing front-end.

The systems discussed above are designed for compiler or environment construction. There is another category of systems that have been primarily used for tool construction. REFINE [REASONING SYSTEMS, INC of Palo Alto CA 1989] and SCRUPLE [Paul and Prakash 1994] provide a (tree-based) pattern-action language to locate interesting pieces of the ASG and perform operations on it. REFINE includes procedural constructs as well; it is an extremely powerful analysis framework. In particular REFINE allows transformations, whereas GENOA does not. While GENOA is not as powerful, it's easier to learn, and quite efficient in practice;

in addition the simple procedural GENOA language allows the querying mechanism to be retargetable. SCRUPLE uses an optimized finite-state machine based execution mechanism to gain high levels of efficiency. The same authors have also discussed a theoretical algebraic model for querying parse trees [Paul and Prakash 1996]. There are also several tools that use an `awk`-like language for operations on parse trees. These systems can leverage the widespread user-base and knowledge about `awk`. A* [Ladd and Ramming 1995] can be used to specify patterns on parse trees; the pattern-based language used in A* is almost exactly just the `awk` language. TAWK [Griswold et al. 1996] is a similar system also relies on patterns on parse trees to implement syntactic analyzers; however, it addresses several key issues raised by previous systems such as SCRUPLE and A*. Most notably: TAWK allows patterns based on abstract syntax, rather than concrete; it allows the toolsmith to select the granularity of the analyzed code (at the level of *statement, function, file* etc.); actions associated with patterns can be defined in the familiar C language. TAWK aims to achieve improvements in both memory-usage efficiency and ease-of-use. Ponder [Griswold and Atkinson 1995] is a complete *framework* for the construction of source-code analysis tools. Ponder is designed as a layered architecture with several elements for parsing, parse tree representation, and parse tree querying. As in the case of TAWK, efficiency (*e.g.*, the compact representation of parse trees) and ease of use are key goals. A vital component of Ponder is a simple parse tree-oriented scripting language designed for toolsmiths to encode analysis tasks. Ponder has been tested on the MUMPS language, and can be re-targeted for C. All these systems, unlike GENOA, *require the construction of a specialized parser (and/or front-end) for each new language/dialect.* Pattern-oriented languages can be quite convenient for simple queries. However, as pointed out in [Crew 1997], languages such as GENOA which offer procedural abstraction can provide higher levels of reuse, which is quite useful for more complex analysis tasks. Crew [Crew 1997] has created a Prolog-like language, called ASTLOG for querying parse trees. This language makes elegant of use of unification and backtracking to allow compact specification of analysis tools. This tool was attached to a proprietary C++ parser through manual effort; however the author has ([Crew 1997], Section 5, "Future Work") stated an intention to using GENOA-like technology to make the tool more easily retargetable.

The LSME [Murphy and Notkin 1996] system uses a different approach to building source code analyzers; it abandons parsing altogether! Analyses of source code are expressed as regular expressions of lexical tokens. For example, forward function declarations can be described by the pattern:

```
[ type functionName '(' type+ ')' ';' ]
```

This pattern searches for an occurrence of a type token (`int`, `char` etc.) followed by a function name, followed by one or more type tokens within parantheses. The key idea in LSME is the possibility of expressing complex analysis tasks as regular expressions of lexical tokens, and detecting pattern occurrences *without* resorting to parsing. This approach has several advantages. LSME is language independent. Analysis tools implemented with LSME are tolerant of syntax errors, linguistically variant dialects (such as K&R C and ANSI C) and can be used with inchoate source code. The authors of [Murphy and Notkin 1996] report a very efficient implementation: LSME specifications are compiled into highly optimized finite state machines. While LSME provides the ability to quickly create "approximate" analysis tools, there are some complications: it can be difficult to implement accurate analyses for complex languages like C++. For example, in the case of C++ call-graph analysis (See Example 4, page 14), recognizing overloaded operator calls would be non-trivial. GENOA is based on parsing and ASG's. Thus, it can be used to build more accurate tools than LSME; on the other hand, GENOA-based tools are not forgiving of syntactically malformed (or dialectically variant) code.

There are some systems [Linton 1984; Horwitz and Teitelbaum 1986; Horwitz 1990] that address both of the issues raised above, *viz.,* retargetable front-ends and flexible querying. In addition to separating the front-end and the back-end, they also provide a compact and convenient language for querying ASG's. The OMEGA system of Linton [Linton 1984] and Horwitz and Teitelbaum's relational-based editing environments [Horwitz and Teitelbaum 1986; Horwitz 1990] are examples. Linton builds a front-end for ADA that compiles programs directly into a set of predefined relations, which are entered into a commercial relational database, INGRES. All editing operations then involve queries and updates to the database. Since the contents of the database can be queried with a relational query language, a range of analyses can be obtained. The advantage of this approach is that instead of re-constructing a front-end, one can simply modify an existing front end to write the ASG into a relational database.

The problem with this approach is that even a simple interactive editing operation, like listing a 10-line file, involve several database operations and is therefore very slow [Linton 1984]. Even if this approach were only used off-line for the analysis of completed code, it would still be doing much needless work: in a complex language like C++, a parser would have to fully materialize dozens of different relations into a database, whereas only a limited number of tuples in a few relations might actually be of interest. Also, as Horwitz and Teitelbaum point out in [Horwitz and Teitelbaum 1986] (pp. 585-586), the limited power of relational operators precludes performing several useful kinds of analyses on source code.

Horwitz and Teitelbaum [Horwitz and Teitelbaum 1986] and Horwitz [Horwitz 1990] address the limitations in Linton's work, while retaining the basic relational representation. First, Horwitz [Horwitz 1990] describes the idea of "implicit relations", which are like non-materialized views in databases – these are generated from ASGs only when they are needed. This addresses some of the performance issues in OMEGA, which stores everything into INGRES. Secondly, Horwitz extends the power of the query language by combining it with attribute grammars: tuples can then be inserted into relations by grammar productions, and the computation of attribute values can include evaluation of relational queries (provided this introduces no circular dependencies [Horwitz and Teitelbaum 1986],pp. 587-588). With this extension, it is possible to express more queries than with pure relational algebra. However, Horwitz and Teitelbaum caution us that there are complications in extending relational query languages (page 578):

> "While adding new operators would solve some problems, it would simultaneously introduce new ones: termination of queries might no longer

> be guaranteed, and the efficiency of query evaluation and view updating would undoubtedly increase " [Horwitz and Teitelbaum 1986]

In Horwitz and Teitelbaum's formalism, with no circular dependencies between the attributes and/or the declared relations, the *termination* condition can be assured; however, the complexity of query evaluation in this formalism is not known. Thus, it may not always be easy to estimate the time required to process any particular query. Although this new formalism is an extension of relational algebra, it's unclear how *expressive* it is, *i.e.,* exactly what kinds of queries can be expressed in this formalism. Finally, the approach to defining a new query involves modifying the attribute grammar itself, and perhaps defining new relations. After coding a new query, the attribute grammar would first be checked for any circular dependencies (the checking time can be exponential in the size of the grammar), and then run through a parser-generator to produce a running analyzer. In some sense, to build a new analyzer, one re-validates the grammar and rebuilds the parser. This could be more involved than the approach we suggest in Figure 1 (page 3), where the parser remains fixed, and only the traversal itself is modified by the query. Pragmatically, our approach represents a different implementation trade-off; it offers advantages for some applications.

Consens *et al* [Consens et al. 1992] describe an application where they use Graphlog, a graphical database query language, for querying a Prolog database containing information about software. In their paper, they are concerned with structural design information about software, such as the "use" dependency relationships between modules. They illustrate how Graphlog queries can be used to identify and remove cyclic dependencies. Although it is theoretically possible to use Graphlog to query parse trees, the authors do not discuss this application.

3. GENOA'S VIEW OF ASG'S

The GENOA specification language is based on a particular, simplified view of ASG's (see Figure 3). This figure shows an input C source file, and the corresponding ASG. Note that the ASG is composed of nodes, which are labeled with *types.* These types correspond to the non-terminals of the source language. For example, an ASG for a C program can have nodes with types such as `Function`, `Assignment`, etc. Each node (based on its type) has one or more *slots*, whose fillers are nodes of particular types, as determined by the syntax and semantics of the source language. For example, in C, a node of type `Assignment` has two slots, `lhs` and `rhs`, both of which are filled by nodes of type `Expression`. Slots may have just one node as a filler, or a list of nodes; for example, the *parameters* slot of a function is filled by a list of nodes of type *variable.* From now on, we refer to all of the fillers of all the slots of a node collectively as *children*; *descendants* refer to all the nodes in the irreflexive transitive closure of *children.* Some types of nodes can have several *derived node types* . For example, in C, the node type `Statement` has (among others) the derivations `If-stmt, While-stmt, Compound-stmt, Goto-stmt`, etc. Derivations of a node type can themselves have derivations, thus forming a hierarchy of node types. Conversely, we say that `Statement` is the *base* type of `If-stmt, While-stmt`, etc. For any language, there is a set of node types, derivations, slots and fillers.

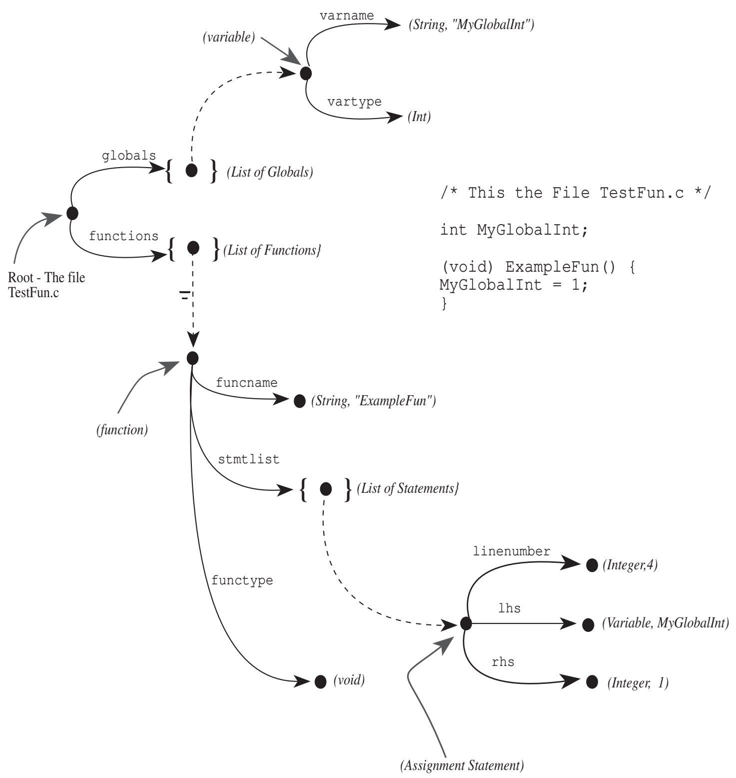

Fig. 3. An Abstract Semantics Graph with typed nodes

A GENOA source analyzer reads the ASG representation of a source program, and can perform a range of traversals, tests, and iterations, and eventually generate output. To implement such analyzers, the GENOA language provides a range different constructs, which we describe in detail in §5. We continue now with a short illustrative example.

4. A SIMPLE EXAMPLE

Without further ado, we present a simple example of a GENOA language specification for a C++ static analysis tool.

EXAMPLE 1. *For each file, print all locations where variables are modified, and where they are just accessed.*

The following is a GENOA language specification of an analyzer that carries out the above task.

```
ROOTPROC VarUse
PROC VarUse
ROOT Cfile;
```

```
{
 [
  (?NameRef
    (IF (AND (HAS-TYPE $parent Assignment) (IS-EQUAL $slot ‘‘lhs’’))
        (THEN (PRINT stdout "Variable %s defined at %s" $token $location))
        (ELSE (PRINT stdout "Name %s accessed at %s" $token $location))))]
}
```

A GENOA language specification is a set of declarations and procedures, some of which are root procedures; these, like the "main" procedure in C, are invoked right after the ASG is built, in the order they occur in the specification. In this case `VarUse` is the (only) root procedure. A procedure in the GENOA language is a series of *constructs*, each of which is an operation on an implicit *current node*. The current node can be accessed by the primitive `$token`; its location is given by the primitive `$location`. A construct may print out the current node, copy it into a variable, move to a child of the current node, etc., but may not modify the ASG. The various types of constructs are syntactically distinguished by enclosing them in different kinds of parentheses.

Returning to the above example, the `ROOT Cfile` declaration specifies that the current node for the initial invocation of `VarUse` is a node of type Cfile, representing the entire file. At this point, we conduct a global search of all the nodes below the current node (`[ ...]` is a global iteration, where a new, local `$token` varies over descendant nodes in a pre-order traversal), looking for nodes of type `NameRef`. (These are essentially references to names of any kind.) When we find name references, we check to see if the parent of that node is of type `Assignment`, and if we got to the `NameRef` node via the `lhs` slot (which means the name reference being assigned to here). If both of these conditions are satisfied, it's a modification; otherwise it's just a use[3]. From this specification, a tool is generated; this tool can be run over each source file using a shell script or makefile, as desired (just like a compiler).

5. THE GENOA LANGUAGE

The GENOA language is designed to implement traversals of ASGs, and to extract information. The GENOA framework represents ASGs as collections of a specific datastructure called a GNODE. Figure 4, page 12 gives the core[4] syntax of the GENOA language. The language consists of *expressions, statements, traversals,* and *declarations*, as shown on numbered lines in Figure 4. These line numbers are used in the discussion below.

Expressions can be built-ins, such as `$token` (line 1), which is the current node, or `$parent`, which is the parent of the current node; `$location` is a string giving the line number and file name of the current node; $slot is a string corresponding to the label on the edge along which the current node was accessed. Expressions can be variables, constants, or booleans (line 2) or list operations (Lines 3 through

[3] This is an obviously over-simplified example, presented for illustrative purposes.

[4] Full details can be found in the manual, available from [Devanbu 1998]; the shortened treatment here omits such details as IF, READ, OPEN/CLOSE (for files), INCLUDE (for C/C++ headers), external (foreign) function declaration and usage, GENOA FUNCTIONS which return values etc.

Expressions

constant	:=	**\$token** $\mid$ **\$parent** $\mid$ **\$location** $\mid$ **\$slot**	1
expr	:=	*Variable* $\mid$ *Constant* $\mid$ *boolean*	2
	:=	**(CONS** *expr expr* **)**	3
	:=	**(APPEND** *expr expr* **)**	4
	:=	**(LENGTH** *expr* **)**	5
	:=	**(CAR** *expr* **)**	6
	:=	**(CDR** *expr* **)**	7
boolean	:=	**(IS-EQ** *expr expr* **)**	9
	:=	**(IS-EQUAL** *expr expr* **)**	10
	:=	**(IS-MEMBER** *expr expr* **)**	11
	:=	**(HAS-TYPE** *expr typetag* **)**	12
	:=	**(IS-NULL** *expr* **)**	13

Statements

stmt	:=	*assign* $\mid$ *print* $\mid$ *eval* $\mid$ *call* $\mid$ *condstmt*	1
assign	:=	**(ASSIGN** *variable expr* **)**	2
print	:=	**(PRINT** *variable string expr** **)**	3
call	:=	**(CALL** *tag expr** **)**	4
eval	:=	**(EVAL** *nodetype string* **)**	5
condstmt	:=	**(COND** *onecond** **)**	6
onecond	:=	((*boolean*) *traversal**)	7

Traversals

traversal	:=	*stmt* $\mid$ *slot-traverse* $\mid$ *test-traverse* $\mid$ *list-traverse* $\mid$ *subtree-traverse* $\mid$ *fulltree-traverse*	1
slot-traverse	:=	⟨*slotname traversal** ⟩	2
test-traverse	:=	(*? typetag traversal**)	3
list-traverse	:=	{*typetag traversal** }	4
subtree-traverse	:=	[*traversal**]	5
fulltree-traverse	:=	[∧ *traversal** ∧]	6

Declarations

declarations	:=	*vardecl* $\mid$ *procdecl*	1
vardecl	:=	*globaldecl* $\mid$ *argdecl* $\mid$ *localdecl*	2
globdecl	:=	*globalvardecl* $\mid$ *globalfiledecl*	3
globalvardecl	:=	⟨newline⟩ **global** *varspec* **;**	4
globalfiledecl	:=	⟨newline ⟩ **file** *tag string* **;**	5
localvardecl	:=	⟨newline⟩ **local** *varspec* **;**	6
argdecl	:=	⟨newline⟩ **arg** *varspec* **;**	7
varspec	:=	*vartype tag* **;**	8
vartype	:=	**node** $\mid$ **float** $\mid$ **string** $\mid$ **int**	9
procdecl	:=	⟨newline⟩ **PROC** *tag* ⟨newline⟩	10
		ROOT *typetag* **;**	11
		[*argdecl* $\mid$ *localvardel*]* ⟨newline⟩	12
		{ *traversal* * }	13

Fig. 4. Syntax of the GENOA query language

7). Booleans can be an equality (line 9), list member operations (line 11), a test if a node is of a certain type (line 12) or a nil list test (line 13).
Statements can be assignments (statements, line 2), prints (line 3), subroutine calls (line 4) or an escape to C: the **eval** operation simply executes the C statement (a string in quotes); this C statement is expected to return a node of the type specified. Macros are provided to translate between GNODE's and C data structures (See [Devanbu 1998] to get a copy of the manual). Conditional statements (lines 6 and 7) are like LISP's.
Traversals form the core of the language. These are the constructs that allow the description of traversals over ASGs. A *typetag* simply corresponds to the name of a node type, such as `Statement`, `Function`, or `Expression`. *Slot-traverse* (line 2) moves the current node to the filler of the slot corresponding to *slotname* (*e.g.,* the `lhs` slot of an `assignment` node on line 16 on page 10), and executes *traversal** over it. (If this is a list, then one might use a list iteration – see below – to traverse each of its elements.) *Test-traverse* (line 3) checks if the current node is of type *typetag*, and if so, executes *traversal** over it. In other words, it is equivalent to

```
(IF (HAS-TYPE  $token typetag)
            (THEN traversal) )
```

List-traverse (line 4) iterates over a list of values (such as the list of variables that form the parameters of a function), executing *traversal** over each one. *Subtree-traverse* (line 5) iterates over all the descendents the current node, and executes *traversal** over each one. A common idiom in the GENOA language is to use *list-traverse* or *subtree-traverse* with an embedded *test-traverse* in order to find nodes of a particularly type.

Finally, *fulltree-traverse* is just a subtree-traverse beginning at the root node. It should be noted here that both *subtree-traverse* and *fulltree-traverse* visit the ASG nodes in pre-order discipline; additionally, the slots are expanded in the inverse order given in the GENII interface specification. However, by combining the list traversal operator with the use of specific slot accessers, the GENOA programmer can create specific traversals to suit his/her needs. The control flow graph example (See Section 5) uses a specialized traversal to perform, in effect, an abstract interpretation of the program to generate a control flow graph.
Declarations are required for local and global variables, parameters, and output files, with a simple syntax (lines 1 through 9). The procedure declaration (lines 10 through 13) specifies a procedure name (line 10), the type of the node at which the procedure will be called (the **ROOT** specification, line 11), and a list of local variable and argument declarations (line 12) followed by a list of traversals to be executed on the root node.

Here is then another example of a GENOA specification:

EXAMPLE 2. *Find any violations of the coding style rule that the* `then` *and* `else` *parts of if-statements must be blocks,* i.e., *statements enclosed in braces.*

```
ROOTPROC badiffind

PROC badiffind
```

```
ROOT CFile;
{
  LOCAL GNODE hasdefault;
  [
   (?If
     < ifTHENbranch
       (COND ((NOT (HAS-TYPE $token Block))
              (PRINT stderr "Line %s then part not a block!" $location)))>
     < ifELSEbranch
       (COND ((NOT (HAS-TYPE $token Block))
              (PRINT stderr "Line %s else part not a block!" $location)))>
   ) ] }
```

We first do a global search for `If` statements; we have to check the *then* and *else* parts of the `If` statement. We use the `<ifTHENbranch ...>` construct to move to the *then* branch. If this branch has a value, then we'll want to check if the filler is a C block, i.e., if it has type `Block`. This is done in the `COND` statement, using `HAS-TYPE`. The *else* branch is handled likewise. Coding style checkers have also been discussed elsewhere [Duby et al. 1992].

As a next example, we present a tool which generates an inheritance hierarchy of a given C++ program; the output is generated in the form of edges in a graph; this can be fed to a graph layout tool to generate a visual graph.

EXAMPLE 3. *Generate an inheritance hierarchy of a C++ program.*

```
ROOTPROC inhier

PROC inhier
ROOT CFile;
{
LOCAL GNODE BASE;
[
 (?ClassDef
        <defname (ASSIGN DERIVED $token)>
        <bases {BaseSpec
                    <defname (PRINT ``%s -> %s'' DERIVED $token)> }>)
] }
```

In this query, we iterate over the entire parse tree, starting at the root node, looking for class definitions. When we find one, we store the name of the class in the variable `DERIVED`; we then iterate over the base classes (via the `bases` slot) of this class and print out edges from the derived class to each of its bases. The next examples shows a simple query to create a normal call graph, but with a happy detail.

EXAMPLE 4. *Generate a callgraph for a C++ program.*

```
ROOTPROC FunWhere
FILE cgr
PROC FunWhere
ROOT CFile;
{
```

```
LOCAL GNODE funid;
LOCAL GNODE callid;
<globals
    {Declaration
     (?FunctionDef
       <defname (ASSIGN funid $token)>
          [(?FunCall
             <callname
              (ASSIGN callid $token) >
              (PRINT stdout " %s-> %s" funid callid)
             )])
     }
    >
}
```

This query looks for function definitions; when it finds one, it saves the name of the function in the variable `funid`. Then it searches for function calls below the function definition; when it finds one, it saves the name of called function in `callid` and generates an edge in the callgraph, printed to `stdout`. Behind the apparent simplicity of this query lurks a very pleasant detail: because GENOA works on the ASG, the node type `FunCall` captures *all* types of function calls: not only normal syntactic function calls, but also implicit constructor and destructor calls and overloaded operator calls. For example, if a function `foo()` uses an expression like `''Dr. '' + ''Smith''`, a line of the following form would be generated:

```
"foo" -> "Tmpstring::operator +"
```

This illustrates two important advantages of basing an analysis tool generation environment on the ASG built with a complete and accurate compiler: first, queries can be compact (we just refer abstractly to `FunCall` to capture all types of calls) and second, to the extent which the semantics of the program (like overloaded calls) are available in the ASG, they can be used for analysis. We now describe a fragment of a traditional, if somewhat more complex tool.

EXAMPLE 5. *Generate a control flow graph of a C/C++ program (a separate one for each function).*

The query (with the GEN++ instantiation of GENOA) to extract flow graphs for C++ programs is about 250 lines long. A complete description of the full details of the tool (which handles "dangling elses", `breaks`, `continues`, `goto`'s, *etc.*) is beyond the scope of this paper; we merely illustrate a simplified portion of it to handle `if` statements with block statements (enclosed in { ...} braces) on the true and else branches, and `switch` statements with { ...} blocks for each case. In the following, we assume that the procedure `blockFlow` handles the control flow (as a recursive co-routine) within a { ...} block statement. We also assume that the local variable `curLoc` has the current location in the source code.

```
ROOTPROC cfgen;
```

```
PROC cfgen
ROOT CFile
{
LOCAL GNODE curLoc;
... lines elided for simplicity ...
(?If
    <ifTbranch
              (PRINT stdout ``%s -> %s [label = true]'' curLoc $location)
              (CALL blockFlow curLoc $token)> /* Handle flow on ``then''
                                                 block */
    <ifFbranch
              (PRINT stdout ``%s -> %s [label = false]'' curLoc $location)
              (CALL blockFlow curLoc $token)> /* Handle flow on ``else''
                                                 block */
)
(?Switch
     <switchbody
         (?Block
             <blockbody {Statement
                         (?Case (PRINT control " %s -> %s; " curLoc
                                $location)
                                <next (CALL blockFlow curLoc $token))>
                         (?Default (PRINT control " %s -> %s; " curLoc
                                $location)
                                <next (CALL blockFlow curLoc $token)>)}>)>)
... lines elided for simplicity ...
}
```

In the fragment handling the `If` type, we examine the true and false branches; for each branch, we generate an appropriately labeled condition flow statement; after this (assuming that each branch is a compound statement) we recursively call `blockFlow` to handle it. We pass in the current location and the compound statement to `blockFlow` so that it can handle the flow within the compound statement and continue the flow to the next statement after the `If` statement. For the `Switch` statement, we generate labels for each `Case` and for the `Default` statement; since we assume that each of these is followed by a block, we call `blockFlow` to handle that. Several more applications (in the areas of software metrics, program analysis, and program dynamics visualization) are described in the paper on Aria [Devanbu et al. 1996] and in papers reported by users of GEN++ [Basili et al. 1995; Mendonça and Karamer 1998; Woods and Quilici 1996; Jerding et al. 1997; Bieman and Kang 1995; Karstu and Ott 1994; Devanbu et al. 1996]. One application, a C++ coding standards checker, has been used by several large industrial projects.

The GENOA language has several limitations. First, lists are the only type of data structure available. If other datastructures (*e.g.,* hash tables) are desired, it is necessary to use the foreign function interface. For example, an efficient data flow analyzer [Steensgaard 1996] may require a union/find datastructure. For reasons discussed in the following section, the GENOA language contains no primitives for dealing with transformations on parse trees. In addition, queries work on one file at a time, thus requiring a separate post-processing phase for global information. The tools that are easiest to implement with GENOA are ones where the result can be computed in a single pre-order scan of the parse tree. While we have implemented many tools that don't fall into this category, these typically require more complex coding, or foreign functions for data structures, or both.

Finally, the initial version of the GENOA language did not support recursive functions and functions that returned values (a restriction that has been removed). Without recursion, it would be difficult, for example to write an expression evaluator.

6. FRONT END RETARGETABILITY: THE GENII SUBSYSTEM

GENOA insulates itself from the vagaries of the ASG implementation by assuming an abstract data-type corresponding to a language-independent ASG. This abstract data-type forms a "movable fire-wall" between the backend and the (arbitrary) front-end.

In GENOA, the "fire-wall" is an abstract data-type layer that defines a notion of trees having typed nodes, and operations and looping constructs designed for such trees. This fire-wall is made "movable" by a translator-generator tool which accepts a specification of a target source language $\mathcal{SL}$ and the description of the ASG built by some existing front end (for the language $\mathcal{SL}$), and produces interface routines which can translate the tree built by the front end into the data structures used by GENOA. This translator generator is called GENII (for GENoa Interface Implementor). In this section, we describe how GENOA gets integrated with a front end. Figure 5 illustrates how GENII and GENOA interact. Instantiating GENOA for a new language is accomplished by writing a specification in the GENII specification language, and running this through the GENII applications generator; this creates a set of routines which implement the decorated abstract syntax-tree abstract data-type. We first present an overview of the GENII support for interfacing to front end data structures; and then we show how everything fits together, using Figure 5.

6.1 Interfacing to the ASG data structures

To illustrate the GENII specification language, we show some details of the specification that implements[5] the interface between GENOA and the ASG data structures of the CIN C interpreter [Kowalski et al. 1991]. The full specification is quite long, so we show here only the part having to do with the different kinds of statements in C, and the details of the compound statement. The specification consists of a series of declarations of *node types*. There are essentially three major kinds of node type declarations: a *base node type* declaration, a *node derivations* declaration, and a *derived node type* declaration. The following is a base node type declaration, describing its slots and fillers.

```
Statement:  < "Snode *" > {
                              LineNo:  an Integer
                                         < " host_in ->CValue.LinenoOffset" >
                              Filename:a String
```

[5]The details shown here differ slightly from the actual implementation: the names of the data structures and fields have been made more readable.

```
                                         < " host_in ->CValue.LinenoFname" >
3 }
```

This declares a node of type `Statement`. It is represented in the CIN interpreter by a pointer to the `Snode` data structure. It has two slots, capturing the line number of the file where this statement occurs. The first slot is `LineNo`, with a filler of type `Integer`, and the second is `FileName`, with a filler of type `String`. The fragments of C code in quotes provides the front-end code to be invoked to get the line number of a statement from the data structures used by the front end. The variable `host_in` is *always* implicitly bound to the parse tree node representing the Statement, i.e., it points to an `Snode`. Thus, the fragment on line 1 would return an integer representing the line number, and that on line 2 would return a string representing the filename. These slots would be "inherited" by the various node types corresponding to the different kinds of statements in C. These are specified in a *node derivations* declaration.

```
0 Statement:  [
1               Exit          |
2               Freturn       |
3               Goto          |
4               Continue      |
5               CompoundStmt  |
6               Switch        |
7               Whileloop     |
8               If            |
9               Forloop       |
10              Doloop        |
11              ExprStmt
 ]
```

Line 0 identifies the node type (here it is a `Statement`) for which the derivations are being identified. Eleven different kinds of statements in C are displayed, from `Exit` (line 1) through to `ExprStmt` (line 11) (which is an expression statement, like an assignment statement). We call each of these a *derived node type* in GENII. Clearly, when the front end produces nodes (in its representation of the ASG) corresponding to these different kinds of statements, we need to be able to identify the type of statement it is, and translate it back into a node of the right derived type for GENOA. A derived node type declaration follows:

```
CompoundStmt :  "((CN *) host_in)->CnWhat == NtBlock" {

        LocalVars:  ListOf Variable
                    < "( (CN) host_in)->NtVar"
                    "( (ID) host_in) ->NextVar"
                    >
        StmtList:   ListOf Statement
                    < "(FirstStmt ((CN) host_in))"
                    "( (CN) host_in)->NextStmt"
                    >
}
```

We declare node of type `CompoundStmt` (line 0). The code fragment in quotes is the front end code that is invoked to test if a node is a compound statement. If so, such a node has two slots: `LocalVars` (the filler is a list of nodes, each of which is of type `Variable`), and `StmtList` (here the filler is a list of `Statements`). The code fragments on lines 3 and 4 provide an iterator over the list `LocalVars`, describing how to get the first `Variable` (line 3) and successive variable values (line 4). Note that the two type casts are different: the first type is the compound statement, the second an element in the list of local variables. When the second expression (line 4) returns NULL, the iteration is complete. Similarly, lines 7 and 8 provide an iterator for `StmtList`. Since GENOA can deal with ASG's, occasionally a link can be "backward", i.e., it can point to a parent node. For example, a recursive function can include a function call node which points back to the function definition. These slots are included as a convenience for the toolsmith, but can lead to termination problems while doing a full traversal of the ASG. To avoid this, some slots in the GENII specification can be flagged by preceding them with a "!" character. This excludes these slots from being traversed during a `''[ ... ]''` tree traversal.

Note that there is a crucial limitation of GENOA and GENII: all the accesses to the ASG built by the front end are *read-only.* There is no provision in the GENII interface for code fragments to modify the ASG. GENOA is mainly intended to implement code analyzers, so it is not possible, as in the case of REFINE [REASONING SYSTEMS, INC of Palo Alto CA 1989], to implement transformations.

What are the front-ends to which GENII can attach? Different front-ends may produce ASG's of varying levels of detail: for sufficiently simple languages, or where efficiency is not a concern, it may be possible to generate code on the fly during parsing, without building any intermediate representations. Compilers of more complex languages, like C++ and ADA, need to store a fair bit of information about the source code for symbol resolution, type inference, higher order component instantiation, optimization, etc. GENOA and GENII do not require a full parse tree representing the entire program; whatever is available can be described in a GENII interface specification, and queried with GENOA. However, GENII assumes that the data structures have a certain abstract structure. The assumptions made by GENII are perhaps best clarified with a description of the signature assumed by GENII. A formal notion of an ASG as envisioned by GENII is a quintuple $\prec \nu, \tau, \theta, \sigma, \phi, \succ$ where

—ν is an (arbitrary) set of nodes,

—τ is a set of node types[6].

—θ is a set of boolean type functions $theta_1, theta_2, \ldots, theta_{|\tau|}$ (one for each node type) of type $\nu \rightarrow \{true, false\}$, which returns true just when a node is of the corresponding type.

—σ is a set of slot labels. These are names such *lhs, location, type*, etc which essentially refer to labels on the outgoing edges from nodes on the ASG.

—ϕ is a set of slot functions $(\tau \times \sigma) \rightarrow (\nu \rightarrow \{2^{\nu} \bigcup \{\perp\}\})$ Given a node type, and a slot label, the slot function ϕ returns a function that maps a node to a node,

[6]The set of types correspond to the various constructs in the programming language, such as *multiply expressions*, *if-statements*, etc. There may be a type hierarchy.

or set of nodes that is the *filler* of this slot, or returns $\perp$, or "bottom", which is essentially an error condition[7]

A GENII specification instantiates the elements of this signature with respect to a specific language, and supplies the implementation in a specific front end via the code fragments. Specifically, a GENII specification supplies the following:

(1) A way of instantiating the set of nodes ν (with a front-end that can build a ASG).

(2) Define the different types of nodes (the types of nodes τ that occur in an ASG) *e.g., function declarations, statements, expressions*, etc. These are specified in node type declarations of GENII.

(3) A way of testing each node to determine it's type, i.e., the function θ. For example, given a node representing one of the statements in the body of a function, we may want to know what type of statement it is, *i.e.*, is it a *function call*? an *if statement*? a *do loop*? This is specified in GENII by the code fragments in derived node type declarations such as the one for `CompoundStmt` shown above.

(4) A list of allowable edge labels, or slot names. Such as *lhs, rhs, of-type, formal arguments, function body,* etc. These are specified as slots in the node type declarations.

(5) Given a node, a node type, and a slot label, define function(s) that can find a filler of that particular slot of that node, (if applicable). These are specified in GENII as code fragments that can be used to compute the fillers of given slots for given nodes.

Although we would certainly not claim that the above signature captures all ASG's, in practice it has proven to be quite flexible. To date, GENII has been used to attach GENOA to several different front ends. The full interface specification for the CIN front end to C has declarations of about 90 node types (both derived and base) and is about 800 lines of GENII code. The node types are mostly independent of the kind of front end used, and have to do more with the language; the embedded code fragments are of course specific to the front end being accessed. The specification in this case expands to over 17,000 lines of C++ interface code. There is also a widely distributed implementation of GENOA for C++, called GEN++, which can be freely downloaded [Devanbu 1998]. This interface specification is considerably more complicated: it comprises 290 node types; the GENII specification is about 1600 lines long, and expands to about 37,000 lines of C++ code. In terms of lines of code, the GENII specification is an order of magnitude smaller than the parser. Also, as shown above (in the examples at the beginning of this section) the GENII is a description of the data structures, and is conceptually far simpler than the intricate parsing, scoping and type checking mechanisms in the front end.

[7]Normally, a set of slots is associated with a given type of a node. Thus, a node of type *equality condition* (corresponding, to say, "`a == b`") will have two slots, *lhs* and *rhs*. Now, if we have a node n of type, say *function_call*, and try to take the *lhs* slot, we will get an error; in this case, ϕ *(function_call , lhs) (n)* $= \perp$

There is also a version of GENOA, called ARIA [Devanbu et al. 1996], that has been interfaced to REPRISE [Rosenblum and Wolf 1991], a persistent ASG representation of C++. GENOA has also been attached to the EDG C++ parser, a commercially available C++ parser from Edison Design Group. These efforts demonstrated the practical viability of this approach. For example, in the case of ARIA, the GENII specification to interface GENOA to REPRISE was written by a student (who was initially unfamiliar with both these systems) during part of a summer. Of the four different front-ends (CIN, Cfront, REPRISE and EDG) that we have worked with, none took more than two months to interface to GENOA. This represents significantly less time and effort than would be required to write a complete, fully validated C++ parser and type-checker from scratch, even using sophisticated compiler construction tools; C++ incorporates many intricate details involving features such as inheritance, virtual functions, constructors, destructors, over-loaded operators (and their interaction) that would necessitate a great deal of special-case handling.

Indeed, the intricacy and complexity of the C++ language was the primary motivation to use C++ front-ends in conjunction with GENOA. The Cfront version was the first; Reprise was the next, to provide a persistent parse tree representation; and the EDG interface was done most recently, as the popularity of this particular parser has increased. In each case, the quality, features and linguistic coverage of the respective parser is fully available to the tools that were built with the corresponding GENOA port. Thus for example, the Reprise parser creates a persistent repository of parse trees; tools based on the Aria [Devanbu et al. 1996] port can exploit this persistent store to avoid needless re-parsing. The EDG parser is written in a highly portable dialect of C, and thus that retargeting of GENOA is highly portable as well (GENOA and GENII are both ANSI and K&R C compliant). After the first attachment, to Cfront, was complete, the node type declarations could be used mostly as is for the other parsers; typically just the code fragments had to be changed. In each of our ports, however, we had either complete documentation about, or access an expert on, the internal representation used by the front end. Without this, the retargeting effort would be compounded by effort to learn those details.

There are, of course, limitations to this approach of reusing an ASG built by an existing parser. First of all, in order to write the GENII specification, it is necessary to have a good understanding of the ASG representation used by the front end; however, once this specification is written, the details of the representation are hidden from the toolsmith. The GENII system currently does not support any facilities for constructing or modifying the front-end data structures. This precludes any applications involving code transformation[8]. The GENOA framework currently does not allow for node attribution in the style of attribute grammars [Reps and Teitelbaum 1984; Gray et al. 1992], although this feature could conceivably be added to future versions. In all instantiations of GENOA, it has been attached to

[8]There are some overarching difficulties in transforming C and C++ programs: they generally include some pre-processing directives ("`#ifdef`") and macro expansion. These often do not fit nicely into a ASG representation; this complicates the process of regenerating source code from the parsed representation.

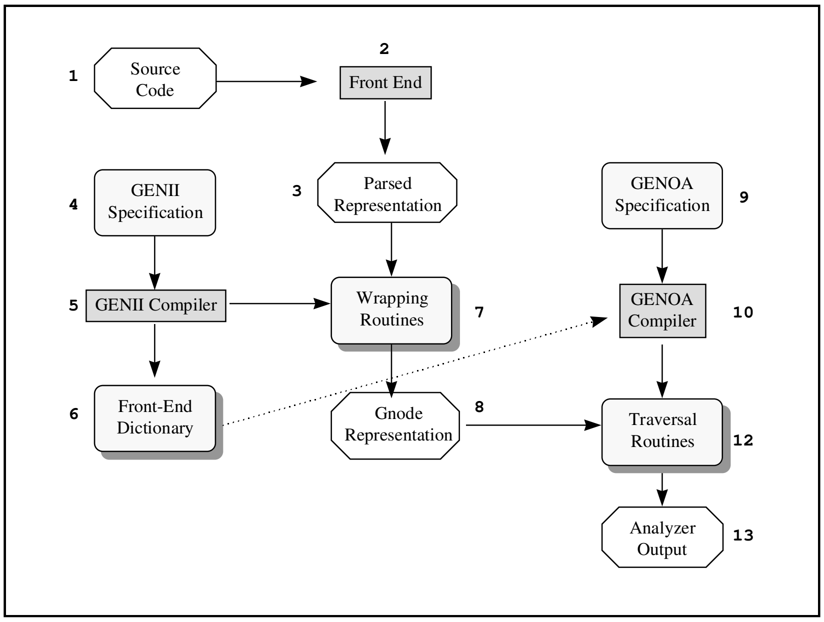

Fig. 5. How GENOA and GENII interact

systems that deal only with "compile-time" information; global information (of the type handled by linkers) is not available. Many kinds of useful analysis tasks involve information at a global level. One approach to address this would be use to GENII to interface GENOA to the data structures built by a symbolic debugger. Another approach would be interface GENOA as a querying mechanism to project management databases [Horowitz and Williamson 1985; Penedo 1986; Lamsveerde et al. 1988]; this would allow access to project management information in addition to source code information. Indeed, the front-end retargeting approach might be a useful way to adapt project management databases to "difficult" languages like C++ and to variant dialects.

Although it has its limitations, the front-end retargeting approach in GENOA has proven useful in practice.

6.2 Implementation Details

Figure 5 shows the top level parts and data flow in the GENOA and GENII systems. A typical analysis tool in the GENOA framework is built with some reused pieces, and with some code generated by specialized compilers. This diagram shows all of the pieces. The dark grey boxes (2,10,5) in the diagram are pre-existing; the specific front-end (2), the GENOA language compiler (10) and the GENII (5) compiler. The data specific to a given source file that the tool processes are shown as octagons: the source code (1), the parsed representation (3) produced by the pre-existing front-end, the GENOA specific GNODE representation (8), and the tool output (13).

The shadowed boxes are artifacts (6,7,12) generated by GENOA or GENII.

To interface GENOA to a given front-end, it is sufficient to write a GENII specification (4) describing the data structures and access methods of the parsed representation produced by the front end (1). From this, the GENII compiler (5) generates wrapping routines (7) which read the representation (3) generated by the front-end (2) and create the wrapped representation (8) as needed; GENII also generates a dictionary (6) used by the GENOA language compiler (10) to validate specifications (9). This validation makes sure that the names of the syntactic (non-terminals & terminals) and semantic (types) entities used in the GENOA specification are appropriate. Once the interface specification (4) is written, a toolsmith can freely write many different specifications (9) to create various tools to process any source files that the front-end (2) can parse. From the tool specification (9), the GENOA compiler(10) generates traversal routines (12) which access the GNODE data structures(8) generated by the wrapping routines (7); these traversal routines also generate the analyzer output (13) particular to the tool specified in (9).

The retargetability hinges on the generated wrapping routines (7) which "lazily" translate the parsed representation (3) into gnode structures (8). If the parsed representation is "in-core", then these routines access the tree directly from the representation created by the front-end. If the representations are persistent, these representations are accessed. These routines are invoked by the tool-specific back-end that is generated from the GENOA specification. In the case when GENOA is used with a parser that is part of an existing *compiler*, the compiler's back-end has to be removed; now the tool-specific "wrapping routines" can be linked to the remaining front-end create the tool executable.

There are 3 main types of wrapping routines: for *traversal, testing*, and *unparsing*. We just describe the traversal routines in detail here. Traversal routines typically take two arguments, a GNODE, and a string representing the name of the slot; they return the filler of that slot. For example, one traversal function might take a gnode representing a `Statement`, and the string `''LineNo''` and return another GNODE representing an integer that is the line number where the statement occurs. We refer the reader back to the GENII specification in § 6.1, page 18. The code for this function (generated by GENII) is shown below, paraphrased for clarity:

```
GNODE Zgna23(GNODE _GNparent) {
      GNODE _GNchild = (GNODE) NULL;
      Snode *host_in;
      int host_out;
      if !(TypeOf(_GNparent,Statement)) {
          ... Error Message!!!  ...
          return _GNchild;
      } else {
          host_in = (Snode *) GETHOSTVAL (_GNparent);
          host_out = (int) host_in->CValue.LinenoOffset
          _GNchild = NEW_GNODE();
          PUTHOSTVAL(_GNchild,Integer,host_out);
          return(_GNchild);
      }
```

```
15 }
```

This generated function `Zgna23` contains some boilerplate with some fragments from the `Statement` specification on page 18. Lines with such fragments are highlighted with "*". A GNODE data structure includes an `enum` field indicating type of front-end AST node it represents, and a pointer to the front end datastructure. The actual types of the front end datastructures are available in the GENII specifications, and are used to generate the local declarations on lines 3 and 4. On line 5, we test the incoming GNODE, `_GNparent`, to make sure it's of the right type, using the `enum` field; the generated dictionary (6, figure 5) is used to also recognize subtypes of `Statement` such as `If`. If `_GNparent`, is not a `Statement`, an error message is generated, and a null output GNODE is generated. Otherwise, the pointer to the AST node representing the statement is pulled out of the `_GNparent` datastructure, and typecast; then the field corresponding to the line number is grabbed. Both of these (lines 9,10) are done using the information from the GENII specification. Now, the linenumber gets stuffed into output GNODE structure, with the appropriate node type (line 12) and returned (line 13). All the traversal routines have a similar structure; the ones that involve slot fillers have lists (like the list of `Statements` in the body of a `CompoundStmt` involve some additional complexity that create a GNODE list. Testing and printing routines involve different boilerplates, but their structure is conceptually similar.

These generated wrapping routines hide the details of the front-end AST implementations; they all deal with GNODES, which form the front-end independent layer in GENOA-based tools. It may be possible in principle to take a legacy parser, and write code manually to adapt all it's data structures to resemble those used by a tool construction framework such as Ponder, A* or Tawk. This would involve identifying each type of datastructure and operations, and "wrapping" each operation within a function so that the result fits within the specific system (Ponder, A*, Tawk etc). There would a lot (very similar) little functions to write. Indeed, the main advantage of GENII is that it eliminates the drudgery of creating wrappers by raising the "level of discourse" for wrapper implementation. One simply describes the nodes that occur in the ASG and their implementation; GENII generates the code to make the ASG look like it was made up of GNODEs. In this way, GENII both simplifies the implementation of the wrappers, and reduces opportunities for mistakes.

6.3 Packaging

Since GENOA is a complex system, comprised of language- and front-end-specific parts and some non-specific parts, the packaging of an instantiation of GENOA (for a given front-end) is clarified here. In the case of GEN++, (with reference to figure 5 the front-end (CFRONT), and the generated wrapping routines (7), and dictionary (6) (for this front-end) are packaged into a library; the GENOA language compiler is shipped as an executable. The package also includes build procedures and a shell script which allows a tool smith to type a single command giving the name of the file that contains the tool specification (9) to build an executable. This script runs the compiler (10), and links the generated traversal routine (12) with the front end (2) and the traversal routines (7) to create the tool. The GENOA framework is equipped to handle both front-ends which can work as subroutines and front-ends that need to be the "`main`" procedure. The GENOA framework allows each generated tool to define it's own command line options and environment variables. However, the handling of these depends on the front-end that is being used. The names and use of such environmental options have to be chosen to avoid interference with the design assumptions in the front end.

7. COMPLEXITY ANALYSIS OF THE GENOA LANGUAGE

Analyzers generated with GENOA are likely to be run over large bodies of code, so it is important to understand their computational properties. In the ensuing discussion, we analyze the complexity of programs written in the GENOA language, with respect to the number of nodes in the input ASG. For example, we can consider the computational complexity of the various operators in the language.

Expressions: The operations listed above (**append, is-equal, cons, is-member,** etc.) are all low-order polynomial time. **cons** of a single node to a list is O(1), and so is **is-equal** with single nodes. **append** of two lists is linear, as is **is-equal** and **is-member**. It should be noted here that the only operations allowed on lists are **cons**, **cdr/car**, and **is-member**.

Traversals: Taking a specific slot of a node (say, the `lhs` of an `assignment` node) is a constant time operation.

Testing if a node is of a node type is O(1). List traversals (again, only over lists of nodes that exist in ASG) and subtree traversals are respectively linear in the size of a list, or of the subtree.

What kinds of queries can be specified in the GENOA language? Clearly, using recursions, one can write a non-terminating computation. Also, using the **eval** construct, one can specify arbitrary programs. Even without recursion and **eval**s, we can readily construct lists that are exponential in the size of an ASG: just embed an **append** of a variable to itself inside a $[\wedge \ \ldots \ \wedge]$ traversal, so that it doubles the length of a list each time. Thus, it is possible to write queries that result in work that is exponential in the size of the ASG. To avoid this, let us restrict one argument of **append** and **cons** expressions to be a tree node, not a variable. Since the tree itself cannot be modified, we can see that in the worst case, we can grow the lists by at most n nodes (where n is the size of the ASG) for each node we visit in the tree; hence, one can only grow lists that are polynomial in the size of the ASG. We call Q_{genoa} the subset of the query language satisfying the above conditions: no recursion, no **eval**, and only one variable allowed as an argument to **append** or **cons**.

LEMMA 1. *Any program written in Q_{genoa} can be executed in time polynomial in n, the number of nodes in the ASG.*

Proof: The proof is by induction. We first analyze the induction step, that gives us a recurrence relation for the complexity of a query of size k in terms of a query of size $k-1$. Once we get this recurrence relation, we can introduce the boundary condition (for a query of size 1), and then derive a closed form.

First, consider an arbitrary query q_{k-1} of size $k-1$, which executes in time $f(n, k-1)$ on a tree with n nodes; we are trying to establish that f is polynomial in n. Assume to begin with that q_{k-1} is memory free (*i.e.*, it has no local or global variables). The most expensive query, q_k, of size k that can be constructed out of q_{k-1} would be to embed it in square brackets, which would execute q_{k-1} over every node in the ASG, n times. Other constructions, such as embedding q_{k-1} in other constructs, or just sequencing $(k-1)$-size constructs would not increase the computational difficulty in this way. Since q_{k-1} is memory free, it takes the same time $f(n, k-1)$ at each node, since it recomputes essentially the same query each time. Thus, q_k would execute in time $n * f(n, k-1)$. If $f(n, k-1)$ is polynomial in n, then so is $n * f(n, k-1)$, giving us the inductive step. Therefore, the worst-case cost of a query of size k to be $n^{k-1} * f(n, 1)$ ($f(n, 1)$ is the highest possible cost of the unit query). Our unit operations (only the expressions, and **print** and **assign** are applicable) are restricted to be at most linear time in the size of the tree. Assuming a constant time t_w (worst-case) for each node, we get the following closed form for the worst-case computational complexity of a query q_k of size k:

$$t_w * n^k$$

This is $O(n^k)$, polynomial in the size of the ASG. Now, we relax the assumption that q_{k-1} is memory free. If q_{k-1} is embedded in square brackets, and if the variables used in q_{k-1} are defined and modified only within q_{k-1} itself (during each evaluation of q_{k-1}), then the variables aren't carried over to next iteration of q_{k-1}; the cost of q_{k-1} is the same for each iteration. The inductive step trivially follows: $f(n, k)$ is simply $n * f(n, k-1)$, and we get the same complexity as before. We also get the same result if the variables are global, and are not modified in q_{k-1}: its cost remains constant.

Finally, we eliminate the assumption that all the variables used in q_k are bound entirely with q_k. In this case, if q_k conducts several iterations of q_{k-1}, the size of the variables used in q_{k-1} can increase with each iteration, and thus, the cost of executing q_{k-1} within q_k can also increase. This complicates the computation of the complexity. To analyze this, let us postulate a size function s, that dynamically measures the size of the global data during the (repeated) execution of the query q, of size k, starting with an initial global data size of σ. Thus, we have

$$s(k, n, \sigma)$$

which gives the worst case bound on the size of the global data in query q_k after it has executed over a parse tree with n nodes. If q_k is repeatedly executed, the size will grow monotonically with each iteration. We now make this claim about s:

LEMMA 2.

$$s(k, n, \sigma) \leq \sigma + c * n^k$$

We omit the proof here for brevity; full details are available in the author's dissertation [Devanbu 1994]. Let us now say that the worst-case complexity of executing a q_k of size k on a parse tree of size n will depend upon the initial size of this global data, and is given by:

$$f(n, k, \sigma)$$

We can now present the complexity result.

LEMMA 3. *For large σ,*

$$f(n,k,\sigma) \leq \sigma * n^{\frac{(k+2)*(k+3)}{2}}$$

This proof is somewhat complex, and we refer the interested reader to [Devanbu 1994] for the full details. The complexity figure is $O(n^{k^2})$, and hence the complexity of a query in Q_{genoa} is polynomial in n, the size of the parse tree, and exponential in the size of the query. This also concludes the proof of Lemma 1. The polynomial time complexity result also indicates that any query expressed in Q_{genoa} can be evaluated "quickly". It provides us with some comfort; we can write various analyzers using this language, and bravely run it over very large collections of source code; they won't take "too long". On the other hand, Lemma 1 puts a limit on the kinds of queries that can be written; one simply cannot express any query that would determine a property that takes, for example, time exponential in the size of the parse tree. This leaves with the question as to what queries *can* in fact be expressed in Q_{genoa}. It may turn out that the queries in this language can be evaluated very fast, but the language itself is so weak that only very few queries can be expressed in it[9]. This motivates us to take a closer 'look at the class of queries that can be expressed in this sublanguage, using techniques developed in database theory.

8. EXPRESSIVE POWER OF GENOA

Database query languages are typically not Turing-complete–they tend be expressively restricted. Queries expressed in relational algebra, with a fixpoint operator, for example, can express *all* polynomial time queries on relational databases, if there is an ordering relationship between the atoms in the domain. We can prove a similar result for Q_{genoa}

LEMMA 4. *Any* PTIME *computation (polynomial in the size of the parse tree) can be expressed in Q_{genoa}*

Proof: The proof[10] is based on a technique due to Immerman [Immerman 1986]. It involves taking an arbitrary Turing Machine that performs a polynomial time computation, and encoding this Turing Machine in Q_{genoa}. If this encoding can always be done, then Q_{genoa} can represent all polynomial-time computations.

So, let us consider an arbitrary polynomial time turing machine $TM^{\mathcal{PT}}$, with an input of length n cells on its tape. Each cell can have symbols σ_i, $i = 1 \ldots m$ and the machine has states s_j, $j = 1 \ldots l+1$, with a distinguished state s_{l+1} denoting acceptance. Since $TM^{\mathcal{PT}}$ is polynomial, we can assume some constant c and integer k such that the machine will always terminate in $c * n^k$ steps.

The encoding of computation of $TM^{\mathcal{PT}}$ is conducted as follows:

(1) First, we encode the tape by three variables: two list variables `rlist, llist` representing the tape on either side of the head, and one variable `now` representing the value under the head. The variable `now` and the elements of `llist` and

[9] Clearly, Q_{genoa} is fairly expressive, since the examples we presented above where all expressible therein.

[10] Full details are in [Devanbu 1994].

`rlist` are one of the symbols $\sigma_i,\ i = 1 \ldots m$. The state of the turing machine is held in a variable `state` which can have just the values $s_j,\ j = 1 \ldots l+1$.

(2) We require that the input tape of $TM^{\mathcal{PT}}$ be presented to our Q_{genoa} program in the form of a tree with $2 * n$ nodes as follows.
   (a) There are two types of nodes, `Integer` and `Cell`. `Cell` has two slots: `value`, which points to a node of type `Integer` giving the value of the node, and `next`, which points to another `Cell` encoding the successor cell.
   (b) We associate an integer value between 1 and m with each of the tape symbols $\sigma_i,\ i = 1 \ldots m$, and encode the input on the tape of $TM^{\mathcal{PT}}$ as a tree using `Cell` and `Integer` nodes to represent the tape as a linked list of nodes.

   This tree encoding of the input forms the input ASG to the Q_{genoa} program which emulates $TM^{\mathcal{PT}}$. For the purposes of simulation, the above special tree-like list is placed as a regular cons-list inside global variable `llist`, by the following Q_{genoa} fragment:

```
[∧ (assign llist (cons $token llist)) ∧]
```

(3) The various steps of $TM^{\mathcal{PT}}$'s computation are simulated as follows.
   (a) A state transition to a state σ is accomplished by simply assigning σ to the variable `state`.
   (b) The movement of $TM^{\mathcal{PT}}$'s head is simulated using **assign**, and the **car**, **cons**, and **cdr** operators of Q_{genoa}. Thus, for example, to move left, we have to 1) **cons** the current value under the head (in the variable `now`) to `rlist`, 2) set `now` to the **car** of `llist`, and 3) set `llist` to the **cdr** of itself. In step 1), we have a complication: Q_{genoa} does not allow two variable operands in a **cons** expression. We solve this problem by having m different **cons** expressions, one for each of the tape symbols $\sigma_i,\ i = 1 \ldots m$, and selecting the right **cons** expression by an m-way **cond**, which finds which **cons** to execute.
   (c) Each state of the $TM^{\mathcal{PT}}$ is handled by an $l+1$-way **cond** expression which compares the value of the variable `state` with each state value $s_i,\ i = 1 \ldots l+1$. Each of these **cond** cases can have a nested **cond** comparing the value of `now` with each of the m symbols, and taking appropriate action. Halting of the machine is simulated by setting `state` to $l+1$; the **cond** case corresponding to this value of `state` will have no further actions, *i.e.,* once we get into this state, we do nothing more.

(4) Finally, to simulate the Turing Machine moves, we have an $l+1$-way **cond**, with cases corresponding to the states of $TM^{\mathcal{PT}}$. Since the machine is guaranteed to run in PTIME, there exist constant c, and integer k such that machine terminates in time $\leq c * n^k$, when given an input of size n. We then merely nest the finite state machine encoding in $k+1$ levels of the "[$\wedge$ ... $\wedge$]" iteration operator to ensure $c * n^k$ steps of TM machine execution for sufficiently large n.

Putting Lemmas 1 with 4, we get the following tight characterization of Q_{genoa} queries on ASGs:

THEOREM 1. *The queries expressible in Q_{genoa} are precisely the queries computable in PTIME on ASGs.*

What is the *pragmatic* implication of these complexity and expressiveness results? First, this tells us that Q_{genoa} is a very useful language. It has the power to specify almost all practical analyses that can be done on source files. Indeed, most of the examples given in this paper can be done in polynomial time. Examples: 1, 2, 5, and 6 in Sections 1 are linear or low-order polynomial; and modulo function pointers, 3, 4, and 7 are as well. Example 1, in Section 4 as shown, is linear time. Examples 2, 3 and 5 in Section 5 are linear time; and again, modulo function pointers, 4 is as well. There are certainly analysis tasks that are intractable: common examples include many of the optimization tasks in the code generation phase of compilers (such as register allocation and instruction scheduling).

Second, a guarantee can be made that any analyzer resulting from a specification written in Q_{genoa} should run "fairly quickly" (*i.e.,* in time roughly polynomial in the size of the ASG representation of the source file). Finally, it is fairly easy to obtain a reasonable estimate of the computational cost, given a query in Q_{genoa}, by inspecting the level of nesting of the [...] operators. Thus a tool builder can determine whether a tool is going to be quadratic, cubic, etc, in the size of the ASG without much difficulty. This is a direct result of the simplicity and compactness of the language. With a more complex language, perhaps one involving complex, interactive rules, and/or with recursive functions, it would be far more difficult to make such an estimate. In our experience, tools can be run over extremely large source bases, so a good estimate of the running time is very helpful.

9. PERFORMANCE EVALUATION

The performance of a GENOA-based analyzer depends on two factors: the particular front-end that is used to build the parse tree, and the desired analysis task. Tools that can be implemented with a single pass of the source code (like a simple call graph generator) will be faster than a complex application like the control dependence graph analyzer we implemented using the ARIA [Devanbu et al. 1996] instantiation of the GENOA framework.

Consider the case of GEN++, an instantiation of the GENOA framework built around the Cfront parser. Tools built with GEN++ (always) first begin with a full parse and type check of the input program, and then perform their specific analysis task. Table 1 presents a comparison of the rate of "consumption" (thousands of lines/second) of various GEN++ tools (using post-processed lines of C++ code, and time measured by `sys + user` times from the Unix `time` command on a Sparc 20/61, with 96 Meg of main memory). As a baseline for comparison, we also show times (in the first two columns) for the CIA++ [Grass and Chen 1990] system, and the C++ to C translation phase of the Cfront compiler. The next column is the consumption rate for an inheritance graph tool, followed by a control flow graph tool, followed by a tool to generate data for the Chidamber-Kemerer [Chidamber and Kemerer 1994] metrics; the next is a call graph extractor. The final column, included for comparison, is the time to run the grep search `egrep -e '[a-zA-Z]* *([^ )]*)'` over the same source base. This pattern is a crude, highly inaccurate approximation for discovering function calls.

It's not surprising that the C++ to C translator is the slowest. CIA++ is the next slowest, since it gathers information about call graphs, inheritance diagrams, macros, include files, *and* builds a highly optimized compressed database. Our sample C++ source base (based on the sample code for buttons, menus, etc., provided the Interviews [M. J. Linton 1989] distribution) contained proportionately greater header file material (type, class, inline and macro definitions) than executable function definitions. This explains why the inheritance diagram tool is slower than the control flow graph tool (which only looks at non-inlined function code, usually in the `.c` file). The call graph tool shows intermediate performance, since call relationships are found both in the header files (due to in-line functions) and elsewhere. Finally, the crude regular expression approximation for function calls is much faster, reflecting its simplicity and consequent inaccuracy (See Example 4).

We now turn to the *space* usage of GEN++. Let us consider the space used by GEN++ falls into 3 categories: the space used by the front-end (**`Cfront`** in this case) to build the parse tree; the space used by the abstract syntax dictionary (Item 6 in Figure 5), and the space used by GNODEs. The space used by the dictionary is generally a constant, about 70 kilobytes (Kb). The space used by the Cfront parse tree varies. Given a particular source file with about a 1000 lines of C code (about 2500 lines after pre-processing), Cfront's parse tree uses up about 2 Megabytes. The *peak total* space used by a GEN++ application also includes (GNODEs) and varies with the application, and is shown in Table 1. We did not have the ability to build an instrumented version of CIA and Cfront to collect the memory usage data, so that is omitted. In addition, the same source file was processed by A* [Ladd and Ramming 1995] and Tawk [Griswold et al. 1996] to produce a function call graph (we had to use a C program rather than a C++ program). Tawk used about 250Kb, and A* uses about 833Kbytes[11]. Tawk in particular is carefully optimized for low memory usage; it actually adjusts its memory foot print to reflect the particulars of the required analysis task. GEN++ is uses more memory than these other tools. There are two reasons for this: first, the front end used as is, without any modifications or optimizations. GEN++'s front end is a full C++ parser, whereas the others are C parsers. C++ is much more complex than C, and the datastructures that represent the ASG are more cumbersome. Second, the retargeting machinery uses space to provide a "neutral" representation for use by the back-end. So in this sense, GENOA trades off space for retargetability. But with a front end that uses less memory, we could expect GENOA-based tools to have a smaller footprint.

Overall, this table reflects our experience with GEN++ based tools; performance is within the range of custom-implemented tools such as CIA++. Viewed alternatively, the *time* penalty for the extra "parser retargeting" machinery in GEN++-based tools is moderate, and the code generated from compact GENOA language scripts is still tolerably efficient. The data in Table 1 shows that the performance of GENOA-based tools is comparable to tools of similar capability [Ladd and Ramming 1995; Murphy and Notkin 1996; Griswold et al. 1996; Paul and Prakash 1994]. There is an important caveat, however, with GEN++ — it is not incremental. If a

[11]The author gratefully acknowledges J. C. Ramming and W. G. Griswold for generously donating their valuable time to obtain this data.

Tool	CIA++	C++ to C (Cfront)	IS-A Diagram	CFG	Metrics (C-K[12])	Call Graph	Regex Match
kLines/Sec	3.3	2.9	3.9	6.0	4.2	5.5	45.8
Kilobytes	—	—	2920	3493	2908	3014	Minimal

Table 1. Performance: source line consumption rate Comparisons (kLines of pre-processed C++ code per second, measured as `user + sys` by the UNIX^{TM} `time` command.) and memory usage with PURIFY^{TM}.

source file is changed, it would have to be completely reparsed: neither the underlying parser (Cfront) nor GENOA are capable of incrementality. It would be desirable to redesign GENOA (including the query language and the retargeting machinery) to exploit an incremental parser: we intend to pursue this in future work.

10. CONCLUSION

We have described the GENOA framework, which is a portable, language-independent querying mechanism for abstract semantics graphs. We also discussed some the theoretical properties of the GENOA query language. We have used this system to build GEN++, a static analyzer generator for C++. The techniques described here enabled us to implement the GEN++ system very quickly, and at low cost. Perhaps because of the difficulties involved in implementing a C++ parser, at the time of first release (Fall 1993) it was the only available system of its kind for C++. The retargetability of GENOA has been proven in practice by implementing interfaces to four different, independently developed front-ends. It has been used to generate analyzers for a wide range of applications, including metrics, case tools, reverse engineering, testing and coding standards enforcement. Specific applications include a tool to gather the Chidamber/Kemerer object-oriented design metrics for C++ [Chidamber and Kemerer 1994], a slice-based cohesion measurement tool [Karstu and Ott 1994] (which incorporates a static slicer), an architectural recovery tool [Mendonça and Karamer 1998] a control flow graph generator, a control dependence analyzer, and a path condition generator (both reported in [Devanbu et al. 1996]). GEN++ can be freely downloaded [Devanbu 1998]; it has many active users, and is supported by the author.

REFERENCES

BALLANCE, R., GRAHAM, S., AND VANTER, M. V. D. 1990. The PAN language-based editing system for integrated development environments. In *Proceedings of the SIGSOFT Symposium on Software Development Environments* (1990).

BASILI, V., BRIAND, L., AND MELO, W. 1995. Measuring the impact of reuse on quality and productivity in object-oriented systems. Technical Report CS-TR-3395, University of Maryland, Computer Science Department.

BIEMAN, J. AND KANG, B.-K. 1995. Cohesion and reuse in an object oriented system. In *Proceedings Proc. Symposium on Software Reusability (SSR'95)* (1995).

BORRAS, P., CLEMENT, D., DESPEYROUX, T., INCERPI, J., KAHN, G., LANG, B., AND PASUAL, V. 1988. Centaur: The system. In *Proceedings of the Symposium on Software Development Environments* (1988).

CHEN, Y. F. AND RAMAMOORTHY, C. V. 1986. The c information abstractor. In *Proceedings of the Tenth International Computer Software and Applications Conference (COMPSAC)* (Chicago, October 1986).

CHIDAMBER, S. R. AND KEMERER, C. F. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering.*

CLEAVELAND, J. C. AND KINTALA, C. 1988. Tools for building applications generators. *AT&T Technical Journal.*

CONSENS, M., MENDELZON, A., AND RYMAN, A. 1992. Visualizing and querying software structures. In *Proc. of the 14th Int'l Conf. on Software Engineering* (1992). IEEE Press.

CORBI, T. A. 1989. Program understanding: A challenge for the 1990's. *IBM Systems Journal 28*, 2.

CREW, R. F. 1997. ASTLOG: A language for examining abstract syntax trees. In *Proceedings, First Usenix Conference on Domain-Specific Languages* (October 1997).

DEVANBU, P. 1994. *Software Information Systems.* Ph. D. thesis, Rutgers University.

DEVANBU, P. 1998. The GEN++ page. **http://seclab.cs.ucdavis.edu/~devanbu/-genp**.

DEVANBU, P., BRACHMAN, R., SELFRIDGE, P., AND BALLARD, B. Lassie—a knowledge-based software information system. *Communications of the ACM.*

DEVANBU, P., KARSTU, S., MELO, W., AND W.THOMAS. 1996. Analytical and empirical evaluation of software reuse metrics. In *Proceedings, Eighteenth International Conference on Software Engineering* (1996). IEEE Press.

DEVANBU, P., ROSENBLUM, D. S., AND WOLF, A. L. 1996. Generating testing and analysis tools with ARIA. *ACM Transactions on Software Engineering and Methodology.*

DUBY, C. K., MYERS, S., AND REISS, S. 1992. Ccel: A metalanguage for c++. Technical Report CS-92-51, Dept. of Computer Science, Brown Univeristy.

GRASS, J. AND CHEN, Y. F. 1990. The C++ Information Abstractor. In *The Second USENIX C++ Conference* (April 1990).

GRAY, R., HEURING, V., LEVI, S., SLOANE, A., AND WAITE, W. 1992. Eli: A complete, flexible compiler construction system. *Communications of the ACM.*

GRISWOLD, W. G. AND ATKINSON, D. 1995. Managing design trade-offs for a program understanding and transformation tool. *Journal of Systems and Software.*

GRISWOLD, W. G., ATKINSON, D., AND MCCURDY, C. 1996. Fast, flexible, syntactic pattern matching and processing. In *Fourth Workshop on Program Comprehension, Berlin Germany* (1996). IEEE Press.

HABERMANN, N. AND NOTKIN, D. 1986. Gandalf: Software development environments. *IEEE Transactions on Software Engineering 12*, 3 (Dec).

HOROWITZ, E. AND WILLIAMSON, R. 1985. Sodos – a software documentation support environment: Its use. In *Proceedings of the Eigdth International Conference on Software Engineering* (May 1985). IEEE Computer Society.

HORWITZ, S. 1990. Adding relational query facilities to software development environment. *Theoretical Computer Science 73*, 2.

HORWITZ, S. AND TEITELBAUM, T. 1986. Generating editing environments based on relations and attributes. *ACM Transactions on Programming Languages and Systems 8*, 4.

IMMERMAN, N. 1986. Relational queries computable in polynomial time. *Information and Control.*

JERDING, D., STASKO, J., AND BALL, T. 1997. Visualizing message patterns in object-oriented program. In *Proceedings, Nineteenth International Conference on Software Engineering* (1997). IEEE Press.

KARSTU, S. AND OTT, L. 1994. An investigation of the behaviour of slice based cohesion measures. Technical Report CS-TR 94-03, Michigan Technical University.

KOWALSKI, T., SEQUIST, C., ELLIS, B., GOGUEN, H. H., PUTTRESS, J. J., CASTILLO, C. M., ROWLAND, J. R., RATH, C. A., WILSON, J. M., VESONDER, G., AND SCHMIDT, J. L. 1991. A reflective c programming environment. In *Proceedings of the International Workshop on UNIX-Based Software Development Environments* (1991). USENIX.

LADD, D. A. AND RAMMING, J. C. 1995. A*: A language for implementing language processors. *IEEE Transactions on Software Engineering.*

LAMSVEERDE, A. V., DELCOURT, B., DELOR, E., SCHAYES, M.-C., AND CHAMPAGNE, R. 1988. Generic lifecycle support in the alma environment. *IEEE Transactions on Software Engineering*.

LINTON, M. 1984. Implementing relational views of programs. In *Proceedings of the SIGSOFT/SIGPLAN workshop on Practical Software Development Environments* (1984). ACM.

M. J. LINTON, P. R. C., J. M. VLISSIDES. 1989. Composing user interfaces with interviews. *IEEE Computer 22*, 2.

MENDONÇA, N. C. AND KARAMER, J. Eds. 1998. *Proceedings of the Workshop on Program Comprehension* (Los Alamitos, California, April 1998). IEEE Computer Society: IEEE Press. 1996.

MURPHY, G. C. AND NOTKIN, D. 1996. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*.

PAUL, S. AND PRAKASH, A. 1994. A framework for source code analysis using program patterns. *IEEE Transactions on Software Engineering 20*, 6 (June).

PAUL, S. AND PRAKASH, A. 1996. A query algebra for program databases. *IEEE Transactions on Software Engineering 22*, 3.

PENEDO, M. H. 1986. Prototyping a project master database for software engineering environments. In *Proceedings of the SIGSOFT/SIGPLAN workshop on Practical Software Development Environments* (1986). ACM.

REASONING SYSTEMS, INC OF PALO ALTO CA. 1989. *REFINE User's Guide.*

REPS, T. AND TEITELBAUM, T. 1984. The synthesizer generator. In *Proceedings of the Symposium on Software Development Environments* (1984).

ROSENBLUM, D. AND WOLF, A. 1991. Representing Semantically Analyzed C++ Code with Reprise. In *The Third USENIX C++ Conference* (April 1991).

STEENSGAARD, B. 1996. Points-to analysis in almost linear time. In *23rd ACM Symposium on Principles of Programming Languages* (1996). ACM.

STEFFEN, J. 1981. *The CScope Program, Berkeley UNIX Release 3.2.*

STOREY, M.-A., WONG, K., AND MUELLER, H. A. 1997. Rigi: A visualization environment for reverse engineering. In *Proceedings of the 1997 international conference on Software engineering* (1997).

WOODS, S. AND QUILICI, A. 1996. Some experiments toward understanding how program plan recognition algorithms scale. In *Proceedings of the Working Conference on Reverse Engineering* (Monterey, CA, October 1996).

ZEIGLER, S. Comparing development costs of c and ada. `http://sw-eng-falls-church.-va.us/AdaIC/docs/reports/cada/cada_art.html`.