In Proceedings of the Fourteenth International Conference on Software Engineering (ICSE '92), pages 307–319, Melbourne, Australia, May 1992.

GENOA - A Customizable, Language- and Front-End independent Code Analyzer

Premkumar T. Devanbu
AT&T Bell Laboratories, 600 Mountain Ave,
Murray Hill, NJ 07974*
prem@research.att.com

September 30, 1994

Abstract

Programmers working on large software systems spend a great deal of time examining code and trying to understand it. *Code Analysis* tools (e.g., cross referencing tools such as CIA and CSCOPE) can be very helpful in this process. In this paper we describe GENOA, an application generator that can produce a whole range of useful code analysis tools. GENOA is designed to be *language- and front-end independent*; it can be interfaced to any front-end for any language that produces an attributed parse tree, simply by writing an interface specification. While GENOA programs can perform arbitrary analyses on the parse tree, the GENOA language has special, compact iteration operators that are tuned for expressing simple, polynomial time analysis programs; in fact, there is a useful sublanguage of GENOA that can express precisely all (and only) *polynomial time (PTIME)* analysis programs on parse-trees. Thus, we argue that GENOA is a convenient "little language" to implement simple, fast analysis tools. We describe the system, provide several practical examples, and present complexity and expressivity results for the abovementioned sublanguage of GENOA.

1 Introduction

The growing cost of software development, particularly in larger systems, is well documented. A significant portion of this cost is due to the time

*The Author is also with the Department of Computer Science, Rutgers University, New Brunswick, NJ 08903

programmers spend on maintenance. Programmers are constantly trying to comprehend large, unfamiliar pieces of code. In a recent paper on the LaSSIE system [?], 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. Current technology provides *analysis tools* such as CIA, SCOPE [?, ?] etc; these tools perform a partial, focussed scan of the code, and produce predefined reports (perhaps directly into a database). While these tools can be quite useful, the information they extract is quite limited; as we shall see below, there is often a need for information that is syntactically extractable, but is not available from existing analysis tools.

GENOA (GENerator Of Analyzers) is an applications generator that produces arbitrary analyzers from specifications. The GENOA specification language uses the vocabulary of *abstract syntax trees*; it is designed to be programming-language independent. GENOA is intended to work in concert with existing front-ends for programming languages that generate abstract syntax trees. GENII, the companion system, can be used to generate the interface between GENOA and existing language front-ends.

We motivate our work with a simple example of a kind of analysis not performed by most current code analysis tools (for the C programming language [?]), and show how it can be implemented in GENOA. We then describe the details of the GENOA implementation, and describe how GENII can used to hook GENOA up to a front end for a given programming language. We then formally analyze the computational properties of a significant subset of the GENOA query language, which is characterizable as precisely the $PTIME$ computations on parse trees. We then present some more illustrative examples of the use of GENOA. We conclude with a comparison of GENOA with similar systems, and a summary of our research contributions.

2 Some Simple Analyses

Without further ado, we present some examples of analyses for C programs (*i.e.,* examples where C is the *target language*) that can be implemented quickly with GENOA (and are not quite so easy with other systems).

Example 1 *For each source file, print in the report file* `/tmp/globlist`*, all functions where global variables occur on the LHS of assignment statements.*

```
GLOBAL FILEOUT ‘‘/tmp/globlist’’

PROC GlobalFind

ROOTNODETYPE File;
LOCAL GNODE GlobList;
LOCAL STRING TheFuncName;
{
(file
  <globals-
     (assign GlobList $ThisNode) >
  <functions-
    {FunctionDecl
      <funcname- (assign TheFuncName $TheString)>
      [assignment
        <lhs-
            (?variable
               (COND
                 ((MEMBER $ThisNode GlobList)
                  (FPRINT
                     FILEOUT
                     "Function %s modifies variable %s"
                     TheFuncName $TheString)))))>] }>)
}
```

Figure 1: GENOA query- Example 1

While existing tools are able to identify where global variables are used (either read or written into) most of them are unable to detect where they are changed[1].

The GENOA specification to implement this analysis is shown in figure **??**:

We begin our explanation of figure **??** with a description of the GENOA's domain of discourse. The generic abstract data type that GENOA works with is a *node*. Nodes in GENOA are typed; the *types* correspond to terminals (or non-terminals) in the target language grammar (*e.g.*,, `function, statement, expression, variable, constant` etc are considered types of the C programming language). Nodes can have *slots*. Conceptually, there are two kinds of slots: *child*, and *attribute*. Child slots of a node n are generally the {non-}terminals that are generated by a nonterminal of type n in the gram-

[1] In the presence of pointers, detecting exactly all the cases where globals are changed is in general undecidable; we are simply looking for cases where a global variable syntactically occurs directly on the LHS of an assignment statement.

mar of the target language (*e.g.*, a node of type `assignment` in C has two children, `lhs` and `rhs`, both of type `expression`, corresponding to the left and right sides of the assignment statement). Attributes, on the other hand, correspond to properties of nodes (*e.g.,* a node of type `expression` will have an attribute `type-of`, and a `stmt` has an attribute `linenumber`[2].) Section **??** describes how the details of the types, children, and attributes in a programming language are defined in the GENII interface specification language. In addition, GENOA has the generic notion of a list of nodes (*e.g.,* filler of slot `arguments` of a node of type `function` is a list of nodes of type `variable`).

GENOA has several *iteration* operators. There are two kinds of operators - one iterates through all the nodes in a subtree (such as all the nodes below a node of type `function`), and the other iterates through a list (such as a list of nodes of type `variable`). One can also define procedures that can be called on a node. Recursion is also allowed. Essentially, a specification in GENOA defines a traversal of the parse tree in terms of the *types* of nodes encountered, their children, attributes, etc.

Returning to the query in figure **??**, we first set up an output file (line 0). Line 1 names the procedure, `GlobalFind`. Procedures in GENOA always have one implicit argument, the "current node". This node is the *rootnode* of the function. Line 2 specifies the type of the current node to be a node of type "File". We then define a few local variables (lines 3 & 4).

Lines 6 to line 16 form the body of the procedure. First, (line 7) we take the child, `globals`, (the left angular bracket indicates a slot, in this case, a child) of the `file` type rootnode, and store it in variable `GlobList` (line 8). Then we take the child `functions` of the rootnode (line 9). We then iterate through this list (the leftcurly on line 10 is a list-iteration operator) looking qt each node of type `function`. For each function, we save the name of the function (attribute `funcname`, line 11), and then we search the entire subtree of the `function` node, looking for assignment statements (the square bracket with the "assignment" node type, on line 12, indicates a search of the entire subtree rooted at the current node). For each assignment statement node, we take the child `lhs` of this node (13), (the left hand side of the assignment) and check if it is of type `variable` (line 14). We then check to see if the variable (`$ThisNode` on line 15) is a member of the list of globals (16); if so, we print out a message to the effect in the output line (lines 17-21).

[2]We emphasize that the distinction between "child" and "attribute" is only conceptual. Usually, attributes tend to be terminals and children, non-terminals. GENOA doesn't distinguish between them; however, it can be helpful for the GENOA programmer to keep this difference in mind while writing specifications such as the one in figure **??**.

Similarly, other compact GENOA programs can be written to analyze code and answer the following queries:

1. Do any of the routines that call `CollectDigit` directly modify the global variable `CallStatus` (i.e., without following pointers[3].) .

2. Where is the value of the `LampStatus` field of the `StationRec` structure being modified directly?

3. Of the functions that switch on a variable of enumeration type `TrunkType`, which ones handle the `ISDNrateB` case?

4. Is somebody putting a pointer to a `CallRec` data structure directly into a `UserRec` data structure?

5. Do any routines that call the `SendMsg` routine, directly pass an argument of type "pointer to a structure of type `MsgBuf`"?

6. Which routines call *only* the routine `SendMsg`?

7. What functions typecast an expression of defined type `TRUNK_D` to the defined type `PHONE_D` ?

8. Check that no subroutine redeclares a variable in a contained context with the same name as a parameter or a global variable.

The ability to implement the wide range of analyzers illustrated by the above set of questions comes at a price: the structure of GENOA is more complex than existing analysis tools, as we now describe.

3 Structure of Code Analysis Tools

Widely available tools such as CIA, CSCOPE and LINT make a firm, *a priori* commitment to the kind of analysis they perform on the code. CIA, for example, generates a relational database with a predefined schema from C source code. This schema includes objects such as functions, variables, and macros and relationships such as "$\prec$ function *defined-in* file $\succ$" and "$\prec$ function *referenced-in* file $\succ$". This fixed scope is a feature of most analysis

[3]Some of these questions, particularly those where pointers might be involved, cannot be answered purely by static analysis; however, quite often, this is not of concern - direct accesses to variables, structures, etc., without going through pointers, would themselves be useful to find.

tools; this has important implications on their internal design. Analysis tools have a similar structure to compilers; they can include a lexer, a parser and (sometimes even) a type inferencer (which together comprise the so-called *front end*), that build an abstract syntax tree (AST); this is followed by a "data generator". (Instead of generating code from the parse tree, they generate data about the source code.) Since the goal of these tools is fixed and limited *a priori*, it's usually not necessary to have a full syntactic and semantic analysis of the language; they do not need to build a full parse tree. For example, a function-call cross-referencing tool may not do full type checking to ensure that there are no type conflicts. Even compilers very often only build fragmentary parse trees as they scan the input source; after code generation, they discard these partial trees as they scan further.

The designers of source code analysis tools can thus limit the processing to extract only the information needed. There is a good reason for this: writing full, complete, front ends for most languages is quite difficult. There are many tricky details to be considered, even in a relatively simple language like C; languages like C++ and ADA present a far greater challenge.

However, in the case of GENOA, since our goal is to perform *any* analysis requested by our users, we must be able to process the entire language, and extract all the semantic information contained therein into a full AST; a GENOA specification would then simply be translated into a program to traverse this tree in the manner specified. As discussed earlier, implementing a full front end is a daunting task for even one programming language, let alone many different ones.

However, front-ends of most languages have something in common: they all produce data structures that implement an AST. GENOA can use this structure, independent of where it came from. This AST *abstract data type* forms a "movable fire-wall" between GENOA and the front end: we can use attributed trees built by already existing front ends for programming languages. We describe how this is done in the next section.

4 Architecture of the GENOA System

In GENOA, the "fire-wall" is an abstract data type layer that defines a notion of trees with typed nodes, and operations on trees; we call these abstract data types *g-trees*, which are made up of *g-nodes* This firewall is made "movable" by the translator-generator tool GENII, which performs the following function:

- it *accepts* an abstract syntax specification of a target source language $\mathcal{SL}$ and the descriptions of the abstract syntax tree built by an arbitrary front end (for the language $\mathcal{SL}$), and
- it *produces* interface routines that can translate the tree built by the front end into the data structures used by GENOA.

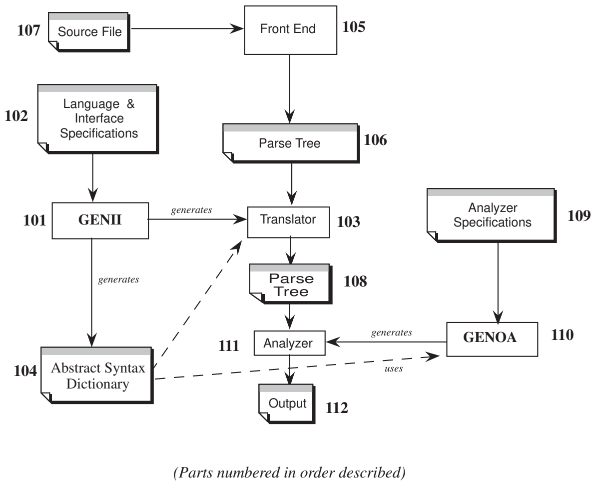

Figure 2: How GENOA and GENII interact

Figure **??** illustrates how GENII and GENOA work together. The shaded boxes denote data items, and the unshaded boxes represent "processing" elements. GENII (101) takes specifications (102) (the GENII specification language is described below) of a source language $\mathcal{SL}$, and the front end, and generates data translation routines (103), as well as an abstract syntax dictionary (104). Analyzers produced by GENOA work as follows. The (existing) front end (105) produces an abstract syntax tree (106) from the $\mathcal{SL}$ source code (107). The (GENII-generated) translation routines (103) translate, on demand, the abstract syntax tree representation (106) into a standard g-tree data structure (108) used by GENOA.

Now, given a specification (109) for an analyzer, GENOA (110) validates it, using the (GENII-generated) abstract syntax dictionary (104), and generates an analyzer (111)). This analyzer runs over the g-tree representation (108) of the source code (107), (invoking the translation routines (103) when needed, to derive the g-trees from the front-end data structures (106)) and produces the desired output (112). It is important to note that except for the one labeled "Analyzer", the other processing elements in the figure remain the same for all analyzers; they depend only on the (existing) front end.

In the following section, we describe how the interface to a given existing front end is specified and generated therefrom.

4.1 GENII - GENOA Interface Usage Specification

In this section, we describe the GENII specification language, which is used to specify the interface between GENOA and a specific front end for a particular language. From this specification, GENII generates the code that implements the language-independent GNODE abstract data type that GENOA uses to implement code analyzers.

The GENII specification language is used to describe the abstract syntax structure of the target language, as well as the data structures used by the particular front end to implement the abstract syntax tree. Space considerations prohibit a description of the full details of the GENII specification language; however, to illustrate the GENII specification language, we show part of the specification that implements the interface between GENOA and the CIN [?] C interpreter[4]. The specification consists of a series of declarations of the *node types* that constitute the abstract syntax of the C language. Examples of node types are non-terminals such as `assignment-statement` and `expression`, as well as terminals such as `identifier`. Each node declaration can contain two kinds of information: first, it describes the schema of the abstract syntax of the language; second, it contains code fragments (particular to the front end) that should be invoked to implement the operations on this node. The full specification for the CIN interface is quite long; in this section, we show the part having to do with the different types of statements in C, and the details of the compound statement.

There are essentially three major kinds of node type declarations; a *single node* declaration, a *node variants* declaration, and a *variant node type* declaration. First, we show a *single node* type of declaration; here we define the children of a single node, and how these children can be generated from the parent single node (by invoking front end code).

[4] CIN is a reflective C programming environment with facilities to create, execute, test, debug, and analyze C programs. CIN has an open architecture that allows it be incorporated into other tools.

```
0 Stmt:  [
1          Exit          |
2          Freturn       |
3          Goto          |
4          Continue      |
5          CompoundStmt  |
6          Switch        |
7          Whileloop     |
8          If            |
9          Forloop       |
10         Doloop        |
11         ExprStmt
12]
```

```
0 Stmt:
1 {
1  LineNo:an Integer
2        < ``((CN *) GvalG )->CValue.LinenoOffset'' >
3  Filename:a String
4          < ``((CN *) GvalG) ->CValue.LinenoFname'' >
5 }
```

This declares a node of type `stmt` to have two attributes - `LineNo`, with a filler of type `Integer`, and `FileName` which is a `String`. They refer to the line number of the file where this statement occurs. The fragments of C in the quotes indicate the front-end code to be invoked to get the line number of a statement from the data structures used by the front end. These attributes would be inherited by nodes of types corresponding to the different kinds of statments in C.
Next, we show how we describe these kinds of statements. This is a *node variants* type of declaration.

Line 0 identifies the node type (here it is a `stmt`) for which the variants are being declared. Eleven different kinds of statements in C are displayed, from `Exit` (line 1) to `ExprStmt` (line 11) (which is an expression statement, like an assignment statement). We call each of these a *variant node* of type `stmt`. Nodes that are of any of these variant node types inherit all the attributes from `Stmt`. Clearly, when the front end produces nodes (in its representation of the parse tree) 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 variant type for GENOA. Here's a variant node declaration:

```
0 CompoundStmt :  "((CN *) GvalG)->CnWhat == NtBlock" {
1
2         LocalVars:  SetOf Variable
3                     < ``( (CN) GvalG)->NtVar''
4                     ``( (ID) GvalG) ->NextVar''
5                     >
6         StmtList:   ListOf Statement
7                     < ``(FirstStmt ((CN) GvalG))''
8                     ``( (CN) GvalG)->NextStmt''
9                     >
10 }
```

We declare a node of type `CompoundStmt` (line 0). The code fragment in quotes is the front end code that can be invoked to test if a node is a compound statement. If so, such a node has two children - `LocalVars` (a list of nodes, each of which is of type `Variable`), and `StmtList`, (a list of `Statements`). The code fragment on line 3 (respectively, 7) tells us how to get the first `Variable` (resp., `Stmt`) in the list of `LocalVars` (resp., `Stmtlist`); the fragment on line 4 (8) tells us how to get the next successive `variable` (resp., `Stmt`) in the list.

The full interface specification for the CIN front end to C has declarations of about 90 node types (both variant and simple); it is about 800 lines of GENII code. The specification in this case expands to over 17,000 lines of interface code that build the data description tables for the attributed syntax tree in CIN, and implement the needed translation routines to translate the CIN data structures to those used by GENOA.

GENII includes facilities to specify unparsing of nodes in the parse tree– *e.g.,* one can specify how the concrete syntax of a node of type `CompoundStmt` is to be derived from the constituents of its abstract syntax specification. Space considerations prohibit a full description of this feature; see [**?**].

As discussed earlier, the interface code generated by GENII helps implement the *g-tree* abstract data type "fire-wall" used by GENOA. The main operations available on a node in a *g-tree* (a *g-node*) are:

- *Extract* the filler(s) of a given slot for a given node (*e.g.,* find the `linenumber` of a `statement` node)

- *Expand* a given node, *i.e.,* find all its children, and

- *TypeOf* a given node: find the type of a given node (*e.g.,* is this *Statement* an `If`, or `Switch`, a `Goto`...?).

As a GENOA program executes, it invokes these operations on the *g-nodes*; the GENII-generated interface code then invokes the appropriate front-end code to perform these operations, and creates other *g-nodes* as needed to store the results. These g-nodes can in turn be expanded, as the analyzer proceeds.

In addition to the interface code, the GENII also generates a set of tables that are used by GENOA to validate an input analysis specification. For example, if a GENOA specification to analyze C source code wants to take the attribute `LocalVars` for a node of type `TypeDef`, GENOA would generate an error message.

Given a particular language and a front end, writing an interface specification in GENII involves two steps. The first, and most significant part of the task is to determine the abstract syntax structure of the language, and the datastructures used by the front end to implement this structure. Given a front end that is designed to be portable, with a well documented abstract syntax implementation, this task is greatly simplified. Once this information has been determined, the next step, coding it into GENII, is quite straightforward. Now, adapting this interface specification to a different front end for the same language, or even to a front end for a slightly different dialect (say from PL/I to PL/C) is a simpler task, since the abstract syntax remains substantially the same; only the code to be invoked in the front end changes, to reflect the different datastructures used[5].

From our experience, we estimate that interfacing to a well-documented front end for a fairly simple, typed, algorithmic language like C or PASCAL would take a few days: far less time than it would take to implement a new front-end from scratch. As the complexity of the target language increases, the abstract syntax structure becomes more intricate; this would call for a longer GENII specification. Generating an interface to a documented front-end for a complex language like C++, PL/I or ADA would be proportionately more time-consuming. A rough rule-of-thumb would be that the time to write a GENII specification for an interface to a front end for a given language grows linearly with the size of the BNF specification of the grammar of the language. Of course, once the GENII specification for a particular front end is done, GENOA can be used to build any number of analyzers; thus, the GENII/GENOA combination provides an economical

[5]If the new front end is missing some pieces of the abstract syntax structure, or adds some new information, the GENII specification would have to be modified to reflect these variations

way to add powerful analysis facilities to existing language front-ends.

5 Complexity of GENOA as a Query Language

Analyzers generated by GENOA are likely to be run over large bodies of code, just as database queries are run over large bodies of data. Therefore, the complexity concerns that exist for database query languages are applicable to the GENOA language. Consequently, we would like to analyze GENOA *qua* query language, and, if possible, identify a subset that can express most practical queries (for example, the ones in section **??**), and still has desirable computational properties. To do this, we consider the various constructs in GENOA, and eliminate the ones that are not absolutely necessary to express the practical queries.

Computationally, there are two main categories of constructs in the GENOA language: expressions and traversals. In the case of *expressions*, the operations listed in Figure **??** - *append, equal, cons, member,* etc., are all low order polynomial time. *Cons* of a single node to a list is O(1), as is *equal* with single nodes. *Append* of two lists is linear, as are *equal* and *member*. Considering *traversals*, taking a specific child of a node (say, the `lhs` of an `assignment` node) is a constant time operation; finding a child of a certain type could be linear in the number of subnodes of a given node. List traversals and subtree traversals are respectively linear in the size of the list, or of the subtree. The full tree traversal ("`[`") iterates over the entire tree, starting with the root.

What class of queries can be specified in the full GENOA query language? Clearly, using recursions, one can write a non-terminating computation; but most practical examples we came across, including the ones listed in section **??**, can be handled without recursion, or for that matter, even procedure calls. Let us therefore omit these from consideration. Next, it is easy to see that with `append`, we can easily construct lists that are exponential in the size of a parse tree (simply embed an append expression inside a "[" traversal, doubling the length of a list each time through). Thus, it is possible write queries whose execution time can be exponential in the size of the parse tree. To eliminate this, we can restrict one argument of append and cons expressions to be elements of the tree, not a variable; this restriction does not affect our ability to write many practical queries (such as the ones in section **??**). With this restriction, in the worst case, we can grow the lists at most n nodes (where n is the size of the parse tree) for each node we visit

Expressions

constant	:=	**$ThisNode** \| **$Frontier**
expr	:=	*Variable* | *Constant*
	:=	(**cons** *expr expr*)
	:=	(**append** *expr expr*)
	:=	(**length** *expr*)
Conditional	:=	(**eq** *expr expr*)
	:=	(**equal** *expr expr*)
	:=	(**member-eq** *expr expr*)
	:=	(**member-equal** *expr expr*)
	:=	(**null** *expr*)

Statements

stmt	:=	*assign* | *print* | *eval* | *call* | *condstmt*
assign	:=	(**assign** *variable expr*)
print	:=	(**print** *variable string expr**)
call	:=	(**call** *tag expr**)
eval	:=	(**eval** *string*)
condstmt	:=	(**cond** *onecond**)
onecond	:=	((*conditional*) *traversal**)

Traversals

traversal	:=	*stmt* | *child* | *slot*
		| *test* | *listmembers* | *subtree* | *fulltree*
child	:=	(**typetag** *traversal**)
slot	:=	⟨**slotname** - *traversal** ⟩
test	:=	(? **typetag** *traversal**)
listmembers	:=	{**typetag** *traversal* * }
subtree	:=	[*traversal* *]
fulltree	:=	[**$ROOT** *traversal* *]

Declarations

declarations	:=	*vardecl* | *procdecl*
vardecl	:=	*globaldecl* | *argdecl* | *localdecl*
globdecl	:=	*globalvardecl* | *globalfiledecl*
globalvardecl	:=	⟨newline⟩ **global** *varspec*
globalfiledecl	:=	⟨newline ⟩ **file** *tag string* ;
localvardecl	:=	⟨newline⟩ **local** *varspec*
argdecl	:=	⟨newline⟩ **local** *varspec*
varspec	:=	*vartype tag* ;
vartype	:=	**node** | **float** | **string** | **int**
procdecl	:=	⟨newline⟩ **PROC** *tag* ⟨newline⟩
		rootnodetype *typetag*
		[*argdecl* | *localvardel*]* ⟨newline⟩
		{ *traversal* * }

Complete Query

g-query	:=	*globalfiledecl** *globalvardecl** *procdecl**

Figure 3: Syntax of the GENOA query language

in the tree.

With these limitations, we call the remaining query language Q_{genoa}.

Query Complexity

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

Proof: The proof is by induction on the size of the query. The full proof is available in a longer version of this paper [**?**]. For brevity, we just present an outline:

As a basis, it is evident that any GENOA query of unit size is evaluable in time polynomial in the size of the tree. For the inductive step, 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 construct the most expensive query q_k of size k that can be constructed out of this query, and show that it can be evaluated in time $f(n, k)$; It turns out, this is done by embedding q_{k-1} within "[$ROOT...]" operator. We then show that if $f(n, k-1)$ is polynomial, so is $f(n, k)$. QED

This shows that any query expressed in Q_{genoa} can be evaluated "quickly". It provides us with the comfort that we can write various analyzers using this language, and bravely run them over very large collections of source code; they will run in time polynomial in the number of nodes in the parse tree. Furthermore, the construction of the "most expensive query" used in the proof yields a simple guideline to estimate the expected complexity of a query: a query can be evaluated in time bounded by a polynomial (of the size of the parse tree) whose degree is one greater than the greatest depth of nesting of the " [" operators in the query. Using this rule, a quick rough bound of the computational cost of a Q_{genoa} program can be made.

On the other hand, Lemma **??** limits the kinds of queries that can be written in Q_{genoa}: one simply cannot express any query that would determine a property that takes, for example, exponential time to evaluate. This leaves open the question as to what queries *can* in fact be expressed in Q_{genoa}. It may be possible that while queries in this language can be evaluated very fast, the language itself is so weak that only very few queries can be expressed in this language[6]. 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.

[6]Though, we have reasons to suspect that Q_{genoa} is fairly expressive, since after all, the examples on section **??** where all expressible therein.

Database query languages (typically) are not Turing-complete—they tend be expressively restricted. Relational algebra for example, cannot express *all* polynomial time queries on relational databases. But we can prove a strong result for Q_{genoa}.

Query Expressivity

Lemma 2 *Any* PTIME *computation on a parse tree can be expressed in* Q_{genoa}

Proof: The proof is based on a technique due to Immerman [**?**]. Again, for brevity, we present only an outline here.

Essentially, an arbitrary PTIME Turing machine computation on a parse tree is encoded directly in Q_{genoa}; The encoding is as follows:

1. We encode the tape by two lists (R-list and L-list), each representing the tape on either side of the head;
2. we encode the parse tree on one side of the head; (say the R-list.).
3. we then represent the transition table of the PTIME machine in a series of GENOA COND statements.
4. Reading, writing and moving the tape, respectively, are achieved by CAR, CONS, and CDR on the appropriate list.
5. When a transition to a halting state is made, certain variables are set to "TRUE", and the tape contains the output.
6. The PTIME running of the machine is simulated thus: by definition, there is some k such that the machine runs in time $\leq n^k$, where n is the size of the parse tree. We then merely nest the finite state machine encoding in n levels of the "[$ROOT" iteration operator (which executes the contents over every node of the parse tree) to ensure n^k steps of TM machine execution[7].

Lemmas **??** and **??** together provide following tight characterization of Q_{genoa}:

[7] The GENOA query might run through the "transition loop" a few more times than necessary, but after the simulated Turing Machine halts, the further runs would be ineffectual; we still however, complete the query in time $O(n^k)$, of course.

Theorem *The queries expressible in Q_{genoa} are precisely the queries computable in PTIME on parse trees.*

This is a result with very practical consequences. First, any analyzer built with GENOA, using the restricted Q_{genoa} sublanguage, is guaranteed to run reasonably fast (and can therefore be run safely over large bodies of code). Secondly, Q_{genoa} is expressive enough to implement any analysis task that is computable in time polynomial in the size of the parse tree. Finally, as we described above, we can quickly estimate the expected execution time of a Q_{genoa} program. Q_{genoa} is thus a suitable language for implementing a whole range of simple, useful, source code analysis tools.

6 Related Work

The most widely-used systems similar to GENOA, particularly in the UNIX^{TM} community, are CSCOPE [?] and CIA [?]. These systems are limited in functionality; they cannot questions such as the ones presented in section **??**. Furthermore, they are restricted to one source language: C.

As discussed in Section **??**, to answer arbitrary questions about the structure of the code, it is necessary to make a full parse of the code, and construct a fully attributed parse tree; an analyzer can then walk over this tree and extract the information desired. Tools that can be used for building parsers, tree-builders and tree-manipulators are also potential competitors for GENOA.

The UNIX tools Lex and Yacc are useful for building lexers and parsers, respectively. Yacc can be used to build context-free grammar (CFG) parsers. Most real programming languages are not context free, so Yacc provides "semantic actions" that can be used to check symbol tables etc. More modern tools, such as the Pan system [?], CENTAUR [?], Gandalf [?] , REFINE [?], the Cornell Synthesizer Generator [?] provide an integrated environment to implement syntactic/semantic processing. In addition to a CFG parser-generator (or syntax-directed editor-generator), they provide ways of implementing semantic processing. The Pan system uses a PROLOG-like rule based method for propagating semantics (type information etc); 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); the Synthesizer Generator uses attribute grammars, while GANDALF uses a special-purpose language called ARL to manipulate abstract syntax trees. REFINE provides a parser-generator environment, and a pattern-action mechanism with powerful pattern-matching (based on unification) over the tree structures. The Metaprogramming System of Cameron & Ito [?] also provides constructs, embedded in a PASCAL-like language, to manipulate an intermediate, parsed representation of source code. However, their system is limited to purely syntactic context-free formulations of source languages, which don't handle any semantic information. The system described in their paper cannot handle some of the queries listed above that involve semantic information.

The tools mentioned above could theoretically be useful in building source analyzer generators; however, in practice, they present some difficulties:

- *Front-end Re-implementation:* In order to use these tools, it would be necessary to implement, for every language, a fully functioning front-end. This is quite difficult, even for relatively simple languages like ANSI C. For more complex, irregular languages like C++ or ADA, the task is formidable. Given the range of different languages and dialects that can be used in a large project, it would be desirable to avoid re-implementing front-ends, but rather use a customizable analyzer that could be readily interfaced to available front-ends.

- *Back-end Languages:* The tree-manipulation and traversal languages in most of these systems are full imperative programming languages—VTP in Centaur, and ARL in Gandalf. They define an abstract tree data type, with operations for traversing trees, testing node-types, etc., and provide the usual procedural programming constructs. One writes analyzers essentially by programming in a full programming language; no guarantees can be made about the complexity of analyzers written in ARL, or VTP. Likewise, the pattern-matching language in REFINE provides a powerful unification facility, but this appears to be undecidable—no published results are available about the expressive power of its pattern-matching language. We have adopted a "query language" approach, using a well understood, expressively limited, relatively compact query language; with GENOA, if an analyzer is written strictly using the sublanguage Q_{genoa}, the complexity is guaranteed to be polynomial.

- *Front-end Separability* Some of the tools are integrated environments; they assume all coding activity—entering, modifying, etc, take place within their context. In order to use them, we have to introduce completely new *modus operandi* into an existing large software project. However, it is usually desirable to leave existing processes and tools alone, and use only the "language processing" part of the environment. This is not always possible.

The OMEGA system of Linton [?] and Horwitz and Teitelbaum's relational-based editing environments [?, ?] come closest to implementing a customizable analyzer suitable for Software Information Systems. Linton is interested in editing environments; he 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 using relational algebra, a range of analyses can be obtained. The problem with this approach[8] is that the simplest interactive editing operations, like listing a 10-line file, take several seconds. 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 Horowitz and Teitelbaum point out in [?], (pp 585-586) the limited power of relational operators precludes performing several useful kinds of analyses on source code.

Horwitz and Teitelbaum [?] and Horwitz [?] address some of the limitations in Linton's work, while retaining the basic relational representation. First, Horwitz describes the idea of "implicit relations" [?] which are like non-materialized views in databases—these are generated from parse trees only when they are needed. This addresses some of the performance issues in OMEGA, which relentlessly stores everything into INGRES. Secondly, Horwitz extends the querying power of pure relational algebra by combining it with attribute grammars as follows. Several relations are declared, along with an attribute grammar specification. Tuples can be inserted into relations by a grammar production, and the synthesis of attributes can include relational algebra expressions (provided this introduces no circular dependencies (See [?] pp 587-588). With this extension, it is possible to express more queries than with pure relational algebra, by combining the algebra with attributes in production rules.

There are two major problems with this approach. First, as Horwitz and Teitelbaum point out, there are complications in extending relational query

[8] Besides the fact that it is only available for ADA.

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 decrease " [**?**]

In Horwitz and Teitelbaum's formalism, without any 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, no *a priori* guarantees can be made about the *efficiency* of processing any particular query. With any program written in Q_{genoa}, we know the execution time is polynomial; furthermore, as explained in Section **??**, the expected complexity cost of any given program can easily be bounded.

Secondly, though the new formalism is an extension of relational algebra, its not clear how *expressive* it is. What kinds of queries can be expressed in this formalism? Clearly, with circular dependencies, it is possible to have non-terminating computations. But can all computations on parse trees be expressed? If not, is there a good characterization of what computations are expressible?

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 have to be first checked for any circular dependencies, (which 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 must re-validate the grammar and *rebuild* the parser. This is more complex than our approach, where the parser remains fixed, and only the traversal itself is modified by the query. After all, the difference between analyzers is only in the information they choose to extract from the tree built by the parser; pragmatically, it can be an added complication to have to re-check the grammar and re-build the parser for each new analyzer. However, as in the case of CIA and CSCOPE, their approach could generate more efficient, custom parsers for each analyzer; GENOA, then, represents a *different tradeoff* between efficiency and ease of implementation.

In a paper in this conference [**?**], Consens *et al* 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.

Graphlog is superficially very different from the GENOA language; however, we have found that Graphlog can be used for querying parse tree. Of course, first a parse tree would have to be translated into a Prolog database[9]. We are currently analyzing the exact relationship between Graphlog and GENOA. Theoretical results available on Graphlog indicate that Q_{genoa} is a more powerful query language. However, our experiments in writing practical queries in Graphlog suggest that it has some features that are very convenient for querying parse trees; we are experimenting with adding some features from Graphlog into the GENOA query language.

7 GENOA- more examples

We now present some more examples of analyzers coded in GENOA, drawn from the examples listed in section **??** (all of these have been coded in GENOA, but we show just two for illustration). The first example is query No. 7 in section **??**.

Example 2 *What functions typecast an expression of defined type* `TRUNK_D` *into the defined type* `PHONE_D` *?*

This example clearly illustrates the need for analysis tools to have access to type information; purely syntactic parsers will not provide this information. The GENOA specification to implement this analysis is shown in figure **??**.

As before, we declare the procedure (0), with the root node type to be a `File` (1), and some local variables (2-3). Then we search all the nodes under the `File` node (line 5) for function declaration nodes. Now, we save the name of the function in the variable `TheFuncName` (6), and then search all the nodes below the function declaration node for `typecast` nodes (i.e., for expressions that do a typecast) (7). We now then initialize the `FromName` variable (8). The `typecast` nodes have a `FromType` attribute and a `ToType` attribute. We check to see if the "from" type (9) is a typedef, and if so, save the name of the type in variable `FromName` (10-11); then we check the "to"

[9]The *GENOA/GENII* approach of directly reading the front end's data structures in memory is probably more efficient in practice.

```
PROC TypeCastCheck

ROOTNODETYPE File;
LOCAL STRING FromType;
LOCAL STRING TheFuncName;
{
[FunctionDecl
  <funcname- (assign TheFuncName $String)>
  [typecast
    (assign FromName "")
      <FromType-
        (?TypeDef
           <name (assign FromName $TheString)>
      <ToType-
        (?TypeDef
           <name-
             (cond
               ((and
                  (equal FromName "TRUNK_D")
                  (equal $TheString "PHONE_D")))
                (PRINTF
                  "Fcn %s typecasts TRUNK_D to PHONE_D"
                  TheFuncName))>))>]]
}
```

Figure 4: GENOA query- Example 2

```
0 PROC SwitchExprCheck

1 ROOTNODETYPE File;
2 LOCAL STRING EnumTag;
3 LOCAL STRING TheFuncName;
4 {
5 [FunctionDecl
6   <funcname- (assign TheFuncName $TheString)>
7   [switchstmt
8     (assign EnumTag "")
9     <SwitchExpr
10      (?EnumType
11         <name- (assign EnumTag $TheString) > ) >
12    (cond
13      ((equal EnumTag "TrunkType")
14       <SwitchCases-
15         {Case
16           <label-
17             (identifer
18               (cond
19                 ((equal $TheString "ISDNrateB")
20                  (printf
21                    "Function %s handles ISDNrateB case"
22                    TheFuncName))) }>) ]
23 }
```

Figure 5: GENOA query- Example 3

type (12). If this is a typedef name also (13), we check to see if the name (14) of the "from" is `TRUNK_D` and the the "to" type is `PHONE_D` (lines 15-18) and if so, we print out a message (lines 19-22).

Our next example is query No. 3 in section **??**.

Example 3 *Of the functions that switch on a variable of enumeration type* `TrunkType`*, which ones handle the* `ISDNrateB` *case?*

This example (figure **??**) illustrates how a GENOA query gathers information from one part of the tree and uses it in another. It also illustrates how the level of nesting in the query is used to go down into, and back up out of, a branch of the abstract syntax tree.

The first 6 lines are similar to the previous example, as we find function declarations and save the function name in `TheFuncName`; in (7) we search for switch statements under the function node. We then initialize the `EnumTag` local variable (8); we then go down to the type of expression that this statement is switching on (line 9) to see if it is of type enum (10); if so, we save the name of enumeration type (line 11) in `EnumTag`, and back up to the parent switch statement node (line 12), and if the enum tag is "`TrunkType`" we go through the list of cases,(14) one by one (15), checking to see if one of them is labeled (16) with an identifier `''ISDNrateB''` (17-19), and print out a message to that effect (20-22).

This example illustrates how each level of nesting in a GENOA corresponds with a particular level of the parse tree. On line 7, we are searching for a switch statement node. Once we've found one, on lines 8 through 10, we traverse down from the switch statement node to the node corresponding to the type of the expression being switched on, check to see if it is an enumeration type, and if so, save the enumeration tag (11). The "`>)>`" (11) closes the traversal down into the switch expression begun on line 8, and on line 12, we are back at the node corresponding to the switch statement. Now, we make sure the enumeration type is indeed `TrunkType` we descend into the cases in the switch statement, looking for the `ISDNrateB` label. Thus, we can start at a node, go down branch of the tree rooted at that node, gather some information, come back and use this information in our next traversal. The nesting of the "`[({<`" controls the search.

The nesting levels and different parenthesis in GENOA might seem a bit confusing to the novice at first; but they're fairly easy to get used to, and they are a convenient shorthand for expressing complex tree traversals. To assist the user in writing GENOA specifications, we are designing an an editing environment that will help the user construct queries, by providing automatic indentation and matching facilities for the various tree traversal operators.

8 Conclusion

We first discussed the need for customizable analyzers in large software systems. We then described the GENOA/GENII system for building arbitrary code analyzers for programs implemented in various programming languages. GENII allows GENOA to be interfaced to different language front ends, thus simplifying the task of implementing a customizable analyzer. In addition, we showed that the GENOA language has some useful iteration constructs that are both expressive and easy to evaluate; in fact, there is a useful sublanguage of GENOA that can express precisely all the queries on parse trees computable in polynomial time. The GENOA and GENII applications generators have been built; GENII was used to build an interface to the CIN system. The GENOA system has been tested on a range of different source analysis problems. We are now implementing a GENII interface between GENOA and REPRISE [?], a C++ front-end.

Acknowledgements

GENOA and GENII were both implemented using the METATOOL^{TM} applications generator generator [?, ?]. Thanks to the following people who provided the inspiration and encouragement for the development of GENOA: Ron Brachman, Dave Rosenblum, Alex Wolf, Dave Kristol, and Bruce Ballard. Many thanks also to Tony Bonner, who pointed me to Immerman [?]. Thanks also to Dewayne Perry for comments on an earlier draft of this paper, as well as for coming up with the acronym "GENOA". My deep gratitude also goes to Alex Borgida for his help and guidance.

References

[1] Ballance, R., Graham, S., and Van De Vanter, M., The Pan Language-Based Editing System For Integrated Development Environments, *Proceedings, 4th SIGSOFT Symposium on Software Development Environments*, Irvine, CA. 1990.

[2] Borras, P., Clement, D., Despeyroux, Th., Incerpi, J., Kahn, G., Lang, B., Pasual, V., CENTAUR: The System, *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1988, Boston, Mass.

[3] Cameron, R.D., and Ito, M. R., Grammar-Based definition of Metaprogramming systems, *ACM TOPLAS*, Vol. 6, No. 1, January 1984

[4] Chandra, A. K., Theory of Database Querys, *Proceedings of the Seventh ACM Symposium on Principles of Databases*, Austin, TX. 1988.

[5] Chen, Y. F. and Ramamoorthy, C. V., The C Information Abstractor, *Proceedings of the Tenth International Computer Software and Applications Conference (COMPSAC)*, October 1986, Chicago, IL.

[6] Cleaveland, J. C., and Kintala, C., Tools for building Applications Generators, *AT&T Technical Journal*, July-Aug 1988. AT&T Bell Labs, North Andover, Mass.

[7] METATOOL Specification Driven Tool, System Overview, AT&T Bell Laboratories, June 1990. *Document Number: 193010-211* (Available from R. Tatem, AT&T Bell Labs, North Andover, Mass.)

[8] Consens, M., Mendelzon, A., and Ryman, A. Visualizing and querying software structures, *Proceedings, ICSE-14* (This conference.)

[9] Devanbu, P., Brachman, R., Selfridge, P., and Ballard, B., LaSSIE: A Knowledge-Based Software Information System, *Communications of the ACM*, 34:5, May 1991.

[10] Devanbu, P. GENOA/GENII - A flexible applications generator for code analysis tools, in preparation.

[11] Habermann, N., and Notkin, D., Gandalf: Software Development Environments, *IEEE Transactions on Software Engineering*, SE-12, December 1986.

[12] Horwitz, S., and Teitelbaum, T., Generating Editing Environments Based on Relations and Attributes, *ACM Transactions on Programming Languages and Systems*, 8:4, 1986.

[13] Horwitz, S., Adding Relational Query Facilities to Software Development environments, *Theoretical Computer Science*, 73:2, 1990.

[14] Immerman, Neil. Relational Queries computable in Polynomial Time, *Information and Control*, **68**, pp 86-104

[15] Kowalski, T., Seaquist, C. R., Ellis, B., Goguen, H. H., Puttress, J. J., Castillo, C. M., Rowland, J. R., Rath, C. A., Wilson, J. M., Vesonder, G. Schmidt, J. L., A Reflective C Programming Environment *Proceedings of the International Workshop on UNIX-Based Software Development Environments*, January 16, 1991, USENIX, Dallas, TX..

[16] Linton, M., Implementing Relational Views of Programs, *Proceedings of the SIGSOFT/SIGPLAN workshop on Practical Software Development Environments*, 1984, Pittsburgh, PA.

[17] Refine Users Manual, The Reasoning Systems Corporation, Palo, Alto, CA..

[18] Reps, T., and Teitelbaum, T., The Synthesizer Generator, *Proceedings of the SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, 1984, Pittsburgh, PA.

[19] Ritchie, R., and Kernighan, B., The C programming Language, Prentice-Hall Publishers.

[20] Rosenblum D., and Wolf A., Representing Semantically Analyzed C++ Code with Reprise, *Proceedings of the Third USENIX C++ Conference*, April 1991, Washington, DC.

[21] J.L. Steffen, Interactive examination of a C program with Cscope, Proc. USENIX Assoc. Winter Conference, Jan, 1985.