UNIVERSITY OF CALIFORNIA,
IRVINE

EPITAXIS
A System for Syntactic and Semantic Software Queries using Deductive Retrieval and Symbolic Execution

A dissertation submitted in partial satisfaction
of the requirements for the degree of
Doctor of Philosophy in Computer Science

by

James Benvenuto

2010

The dissertation of James Benvenuto is approved:

Date:_____

Approved:

_____

Dr. Amelia Regan, Chair

_____

Dr. Brian Demsky

_____

Dr. Ian Harris

University Of California, Irvine

2010

"The way to do research is to attack the facts at the point of greatest astonishment."
--Celia Green

Abstract

Modern computer hardware (multi-core, multi gigahertz processors with gigabytes of RAM and terabytes of disk) along with IDEs allows programmers to build computer programs which are bigger and more complex than they can understand or keep in their working memories. Additionally, the problems these programs are designed to model are ever more complicated. Consequently, programs are full of inconsistencies, mistakes, and incompleteness's. These problems are difficult to detect, difficult to locate, and difficult to correct. Often a change is made by a programmer to fix a problem for which understanding all the repercussions of the change is difficult. Consequently, further bugs are introduced into the code base. Because of the pervasiveness of software in society and the potential severity of the consequences of bugs, software developers need

ever better tools to help them understand, navigate, and follow the consequences of their development and maintenance activities.

This dissertation presents a novel framework based on tree/graph searching and parsing, deductive retrieval, dynamic analysis, symbolic execution, aspect oriented programming, and an *open* interpreter to allow a software developer to navigate, locate features, find bugs, and abstract information in software. The system is designed to have a fast *modify-test cycle* such that the programmer can search and test the software as it is being edited without time consuming recompilation, reinstrumenting, or database repopulating each time an edit is made to the code base. The system is language independent, requiring only files to specify the language grammar, control flow graph transformation, and execution semantics. In addition, because of the flexibility and programmability of the system it is an excellent environment to perform further research on program analysis techniques such as dynamic analysis, symbolic execution and abstract interpretation. A prototype system has been built along with data files for the C programming language which demonstrates the feasibility of the system and its ability to scale to "modern-sized" programs.

"A programmer is a machine for turning coffee into software."
-- Adapted from Alfréd Rényi

Dedication

This dissertation is dedicated to the organic coffee growers of Sumatra.

Contents

List of Tables

List of Figures

"Research is the process of going up alleys to see if they are blind."
--Marston Bates

Chapter 1

1. Introduction

Software is more a logical product than a physical product. This has a number of interesting implications. The actual production cost of software is negligible, all the cost and effort is concentrated on development, evolution and maintenance. Because of the stubborn persistence of Moore's law, except in unusual cases, memory capacity and processing speed are not constraints. This leaves managing complexity as the principal bottleneck in software development [128, 137]. So while we have the hardware capacity to build bigger, faster, and more comprehensive systems, we have been accruing complexity, which has outstripped the human capacity to handle this complexity. This has caused a

"software crisis" [58]. Society has come to rely on these large systems because of their value; they have become integrated into the economic and physical infrastructure of our society. Because of this dependence, these systems must be maintained so they can evolve to meet the needs of the growth of society.

Maintaining and evolving these software-intensive systems is expensive and labor-intensive. Numerous sources in the literature have reported on the high cost of software maintenance, ranging from 50 to 70 percent of the cost of the entire project [17, 101, 121, 128]. Software maintenance is largely a manual process, relying on human brain power to identify problems, perform analysis to find a solution, and finally to implement a modification. Human attention span and memory capacity is inadequate to handle the quantity and complexity of software systems [55, 131] with hundreds of thousands to millions of lines of computer code with a rich and detailed interconnectivity of parts [23]. Consequently, better software tools are needed to support software maintainers, so their reach does not exceed their grasp.

This dissertation research concerns the construction of an integrated development environment capable of software query using *open symbolic execution* and *epitaxic deductive retrieval*. The name comes from the following: 1)

retrieval meaning the ability to access information, 2) *deductive* meaning the ability to not just access information that is directly there but also what can logically and structurally be deduced, and 3) *epitaxic* from the Greek *epi* + *taxis* meaning above or upon an orderly arrangement. This reflects the ability of the system to "grow" new information from the structure and align it to existing information. There are four components to this work: an integrated data structure for representing source code called a virtual abstract semantic graph; a unified language for specifying program queries on the data structure; an *open* symbolic interpreter for expanding the range of queries; and a set of built-in queries that includes both syntactic and semantic based queries. These are also the four contributions from this research.

The system will integrate the representation and search of software on four levels: lexical, syntactic, semantic, and abstract. Via a Prolog-like unification search algorithm, source code searching, program query, and interpretation will be expressible using search predicates. The search predicates provide a single, unified language for specifying program queries and analyses. They are capable of expressing not just an elements lexical relationships and syntactic relationships, they will also be able to express the elements semantic and runtime

relationships because there will exist an integrated interpreter driven by a rule based representation of the program's language semantics. Finally, it will also unify both syntactic and semantic queries into a single framework. This system is incremental because it is embedded within a hybrid character/structure editor. As a result, queries can be performed interactively, as code is written. Query is not limited to batch processing of a snapshot of the code, because re-parsing and analysis is done incrementally.

This unified data structure is based on the following key insight into program analysis. Parsing of source code to construct an abstract syntax tree (AST) is a solved problem. It is a simple matter to specify a grammar for a LALR-parser-generator. However, semantic level analysis is an area of active research. These analyses rely on abstract semantic graphs (ASG)[15], or equivalent data structures, which are constructed from ASTs by adding edges to represent semantic information and removing unnecessary syntactic details. Currently, different ASGs are constructed for each type of query or analysis and the tools must be hand built. There is no general approach to specifying the construction of ASGs and many, many ASGs are needed for a full range of analyses.

The solution is to reify virtual ASGs as they are needed using a unification engine, an AST parser that walks and transforms the AST according to prescribed rules, an *open* symbolic interpreter that can feed the search engine, and a structuring memoization system which can accumulate various pieces of extracted information in a structured form that can also be used as a data-base for further or embedded queries. The language for specifying program queries and analyses is based on search predicates and an *open* symbolic interpreter. They represent rules for tree traversal, syntactic and semantic software query, execution of code and so on. The system can be extended and configured simply by adding classes, rules and predicates to handle additional source languages and to perform additional queries. The system can also be extended by adding *before* and *after* methods, and by overriding methods comprising the interpreter. The data structure and search specification language together form the basis for integrating various forms of queries and analysis. This is *Epitaxis.*

Evaluating this research presents challenges because it addresses an area that is very broad; namely, how can we help programmers deal with the complexities of writing software? The solution that will be presented is designing and implementing a declarative language in which programmers can express

constraints and relationships that their software must obey and then have the locations in their code where these do not hold pointed out to them. We select several such queries to demonstrate the range of effectiveness of this solution. These must be complex enough to be difficult to deal with manually on large bodies of software, specific enough to the program under development so that large monolithic systems have not already been build to find these problems, and simple (relative to the complexity of the relationships they are intended to find) to express declaratively so that they will be worth the effort of writing. We will be evaluating this solution on several specific queries. These queries will be selected to engage the system progressively in layers, simpler at first, and then eventually the full set of features. These queries must be such that it is easy to see that this language is expressive and powerful enough to be applied to many such particular problems, which arise for the programmer. They must also work on realistic size bodies of code and produce answers which have a "real-time" response and a low false positive rate. To be usable by a programmer as he is writing code they must be short and straight forward to write, quick to return a response, and give a response which precise enough to be useful.

The set of test queries will be graded. The first element to validate is basic syntactic querying on these structures. The second element to validate is the structuring memoization system. This will be used to build the control flow graph. The third element to validate will be the *open* interpreter. Finally, the overall *epitaxic deductive retrieval* approach will be evaluated by selecting a set of queries, which rely on all the machinery. These will each be described in more detail in Section 5.3.

This research has an empirical quality to it. A powerful query language will be built and the system will be explored to see some of the range of queries that are possible using it. The ultimate validation will depend on how useful and convenient the range of possible queries are as well as the scalability of the system to handle hundreds of thousands lines of code and its real-time response. It should be straightforward to build and run queries while the software is under development and evolution. This body of queries should build upon itself and grow with the complexity of the software under development. Although the focus of this research is on query process and constraints it is expected that the final system will ultimately accommodate a range of other activities such as program refactoring and language translations as well.

Epitaxic deductive retrieval has the potential to form a new basis for program query in the same manner that grammars form the basis for parsing. With the power of this tool, more software maintenance tasks can be supported. The word "maintenance" is derived from the Latin *manu tenere* which literally means to hold in the hand; with this tool, understanding more complex structures and relationships will be within the grasp of a software maintainer.

1.1 Contributions

In this dissertation, we present *Epitaxis*, a prototype system designed to aid the programmer in understanding and modifying software systems. The features of *Epitaxi*s are designed with the constraint that the system works interactively. That is, the programmer can search or test his program, make changes repeatedly within an interactive time scale without lengthy recompilations, reinstrumentations, or repopulating a database. *Epitaxis* is embedded within a structure editor, although the editing features of the system are not part of this dissertation. *Epitaxis* is also designed to be language independent. Language specificity is supplied by several files: a BNF type grammar file, a Prolog like set of rules for creating a control flow graph for the abstract syntax tree, and a set of rules describing how to execute the control flow

graph. *Epitaxis* currently has definitions for the C programming language. Specific research contributions are described below.

1.1.1 Integration

Epitaxis integrates search on a lexical, syntactic, and semantic level. Search on a semantic level is achieved through the integration of search technology and symbolic execution (as a means to expose the execution state to search).

1.1.2 The Parser

A standard LALR(1) parser generator has been enhanced for interactive reparsing. After an initial AST has been constructed, when a line of program text has been modified, only that line needs to be tokenized. The parser quickly reparses the program using just a few pieces of intact AST surrounding the single lines' token stream. The parser accepts both lexical tokens and AST nodes when reparsing.

1.1.3 Prolog Variants

Epitaxis uses a novel object-oriented Prolog variant to perform search and transformations on the syntax tree and the control flow graph. These include dynamic scoped prolog variables, unification by reference, unification with gatherers and generators, hyper-edges, and self-fulfilling assertions.

1.1.4 Syntactic Search

Epitaxis uses a novel syntactic search system.

1.1.5 Semantic Search

Epitaxis extends program query into the semantic realm using symbolic execution and *collect points*.

1.1.6 Open Symbolic Execution

1.1.6.1 Openness

The symbolic execution interpreter is implemented using *open program design* [88, 89]. Functional units within the interpreter are implemented using generic functions allowing the user to participate in the implementation of the system. The generic functions encapsulate interpreter functionality that the user can override and redefine. This enables the system *user* to become a system *implementer* to get additional functionality.

1.1.6.2 Creation of a Semantic Search Space

Symbolic execution is used as a means to transform a control flow graph into a representation of the program execution state tree suitable for semantic query of the program.

1.1.6.3 Stop on Hit Counts

The symbolic execution interpreter can be configured to track the conditional expressions it has seen by path so the interpreter can terminate a particular path execution when the path steps on itself a set number of times. This allows a more sensible way to deal with loop termination and control over how much of the program executes.

1.2 Outline

The remainder of this dissertation is organized as follows: Chapter 2 relates the current work to the research literature. Chapter 3 presents background material related to understanding *Epitaxis*. Chapter 4 describes *Epitaxis*. Chapter 5 presents the findings. Chapter 6 presents conclusions and directions for future research.

"You should try to remember that a dedicated teacher is a valuable messenger from the past, and can be an escort to your future."

--Albert Einstein

Chapter 2

2. Literature Review

Program Query Systems are a type of Program Understanding Systems. Program Understanding Systems include a wide range of functionality. Because program understanding is such a large and complex area systems tend to specialize or at least focus on an area. One area is visualization. Another area is domain knowledge. These systems offer forms of program query but geared to their specific specialty. In the case of visualization systems the results of queries are clickable views. By clicking on elements of these views other queries are

performed. Comprehensive Software Information System (CSIS) [140] is a system designed to represent software highly integrated with its domain. The software is actually written in its domain representation, which automatically generates a code representation. Here the queries are geared towards the application domain. In this case the transformation goes in the reverse direction: from a semantic net representation of domain features to the source code. Another area where program query is embedded is within program analysis. Here the analysis is generally "hardwired" instead of being available in a general-purpose language. This is traditionally done in two areas: 1) in compilers to validate optimizations to code, and 2) to validate the correctness of software. Validating software can be further divided into two areas: 1) model checking and 2) testing.

There is a wide range of program analysis techniques, including control flow analysis, data flow analysis [78, 79, 114], disjointness analysis [84], program dependence graphs, slicing [64, 80, 125, 136, 138, 139], pointer alias analysis [47, 81, 113, 128, 129], type inference [19, 73], abstract interpretation [40, 85], symbolic execution [11, 38, 42, 43, 91, 92, 94] among others. This technology is beginning to migrate into program query systems themselves since much of this information is also useful to the programmer to understand software. Program Query

Systems tend to have a more generic query mechanism designed to help a programmer interactively “discover” facts about their code.

The complexity of program query systems has grown along with the complexity of the programs they query. There are two reasons for this. They both can be traced back to the availability of more memory and more processing speed. As software has become bigger, it generally becomes more complex with more non-locality. This dictates the need for more powerful tools to understand it, hence more powerful query machinery. The same release from memory confines also allows the program query systems themselves to become more complex. This allows for the possibility of a source transformation and a more robust program representation to query. The evolution of program query systems somewhat reflects this.

2.1 Program Query Systems

One of the first and probably still one of the most widely used program query systems is GREP [126]. It has the advantages of working directly on the textual representation of the program hence no transform is needed. This makes it fast and very tolerant to any irregularities within the source code (i.e. there can be syntax errors, incomplete lines of code, the code can be in any partial state of

development, etc). It also requires no understanding of any subtly of the syntax of the language it is querying. This lightens the cognitive burden on the tool user. The user only needs to understand the language of regular expressions. The down side of GREP systems is the limited range of possible queries. They admit no context and cannot take advantage of saving any state in the search. It is not easy for GREP to tell if what it is finding is embedded within a comment or within a piece of code. Because of its lack of context sensitivity GREP results tend to have low precision. There are various dialects of GREP, one is AWK [3]. AWK uses regular expressions to match pieces of source code but in addition allows pieces of C-like code to be executed when regular expressions match allowing AWK to perform transformations or extended queries.

The limits of regular expression (lexically) based searches were addressed with syntactic versions of these tools. Semantic GREP [25] adds a small amount of syntactic understanding to GREP. It knows what some program constructs are such as functions and can limit its lexical search to within these categories. It also has a simple understanding of relationships between these categories, such as being able to find all functions that call some function. The other useful addition

is queries with transitive closure. Semantic GREP is basically lexical search with a mild amount of *ad hoc* syntactic awareness.

Lightweight Source Model Extraction (LSME) [116] fills an interesting "sweet spot". LSME allows patterns to be expressed with a hierarchical regular expression. The system is basically a lexical scanner with a touch of syntactic contextability supplied by the ability to nest the regular expressions. This way no syntactic constraints are placed on the patterns. Attached actions may be executed when a pattern is matched. The system is optimized for flexibility, speed and tolerance over precision.

The next stage of advancement in query languages came when the source code was transformed into an abstract syntax tree (AST). This allows much more syntactic context to be used to constrain search. Of course, this required more powerful processors to do the transform in a timely manner and more memory to hold the transformation. One of the difficulties of syntactic search is that it requires a more complicated language to express. This adds to the cognitive burden of using these systems. SCRUPLE [119, 120] and TAWK [72] are examples of this approach.

SCRUPLE transforms the source code into an AST and converts the query into a finite state machine (FSM). The states of the FSM represent walking the AST and matching pieces of the query to it. SCRUPLE uses the source language augmented with syntactic wildcards to express queries. Using concrete syntax somewhat eases the burden of learning the query language since the programmer is already familiar with it. The wildcards can be given names so what they match can later be referenced. The concrete syntax also causes problems. Queries have to be syntactically correct so often extra patterns have to be inserted between the patterns of interest to glue them together or to deal with finding patterns that can either follow or be nested within another pattern. Because patterns are syntactically based generalizing patterns such as looping constructs which are conceptually similar but syntactically different can be difficult. In order to enable patterns that are order independent the designers introduced sets. This has the possibility of combinatorial explosion in the number of matching possibility. SCRUPLE keeps the entire AST in memory and queries always search the entire tree.

Another AST based example is TAWK [72]. The idea of TAWK is to combine the precision of syntactic matching with the speed and generality of

lexical matching. In addition, allowing limits to the units being searched instead of the entire program increases speed of search. The query language is made more expressive by using abstract instead of concrete syntax. This also helps make the query language more language independent. Like AWK, TAWK also has C as its action language, which gets triggered when patterns match. TAWK also allows pattern abstractions using a macro-like facility. Backtracking is performed to explore all possible alternatives until the entire pattern is matched or exhausted. Like SCRUPLE, a TAWK pattern is parsed and converted into a FSM that walks over the AST. To reduce its memory footprint TAWK throws out its AST after each search. They trade amortizing parsing costs over multiple queries for a smaller memory footprint.

A* [98] is similar to TAWK. It also is a syntactic generalization of AWK. It converts the input source text into an abstract syntax tree. A* programs are interpreted and have the usual AWK pattern-action syntax. The pattern language allows for control in the traversal of the AST. It however it does not support wildcards or pattern variables.

REFINE [26, 96] is a software development system, which includes a parser generator, an object-oriented database, and a query/transformation

system. Software is transformed into the database as an annotated AST by the parser. The query language uses templates with wild cards and variables to match pieces of code. Rules allow the matched pieces to be output to the user or transformed for re-engineering. In all cases there is a distance between the source code and the transformed database. The transformation is a batch process.

GENOA [57] is a framework for generating tools, which process the AST. Tools such as GEN++ and Aria have been built from this framework. The system has to be integrated with a front-end that does the parsing. GENOA reads the AST representation of a source program and using a scripting language can perform a range of traversals, tests, and iterations eventually generating output. The scripting language has constructs for walking the AST and print statements for outputting information once it is found. GENOA is designed to be independent of the structure of the AST.

CodeSurfer™ [7, 8] is a tool designed to provide “fine-grained” inspection to software. It uses static analysis to build a complex representation of the program, which includes a call graph, points-to graph, and dependence graph. Vertices of the graph represent program constructs such as assignment statements, call sites, conditional branches, etc. An edge in the dependence graph

represents either data dependence or control dependence. Queries are implemented as primitive operations on the dependence graph. This graph is stored entirely in memory. One this structure is constructed CodeSurfer™ provides a system of windows with clickable links for viewing and navigating the dependence graph.

ASTLOG [44] is a prolog based query language that allows for a very flexible but structured search control methodology. It eliminates a lot of the *ad hoc* quality in many query languages because of its general declarative prolog like language. Predicates are used to traverse the program's AST, but instead of importing the AST as prolog like facts, in ASTLOG the interpretation of the predicates and queries are modified to be applicable to external objects. A C/C++ front end provides the accessability to the nodes of the AST. Rather than the usual prolog database ASTLOG has a current object. Every query and term being evaluated as a predicate is interpreted as a pattern that may or may not match the current object (instead of the usual prolog system of looking for matches in the database of facts and rules). It's as if every predicate takes on another hidden term, which is the current object. Of all the query systems surveyed ASTLOG comes closest in spirit to the functioning of the query engine in *Epitaxis*. One of

the main differences is that *Epitaxis* has a more intimate awareness of the structure of the AST nodes as well as the node being object-oriented and admits the use of inheritance in matching. ASTLOG uses an `op` predicate to match the "opcode" (meaning type) of the node and a `kid` predicate to select the child node. There is no hierarchical structure to the nodes relating different types of expression operators or different types of iteration constructs. Also, in *Epitaxis*, the current object is explicitly mentioned in the rules. This gives more direct control over the movement around the parse tree as well as allowing multiple uses so there in essence can be multiple current objects, which can be compared or related in some way. ASTLOG has an interesting reflection mechanism, which allows queries themselves to be used as objects to search. While this is a fascinating mechanism it is used as a way to create aggregates of items found in searches. *Epitaxis* does this much more directly.

Smalltalk Open Unification Language (SOUL) [142, 143] is another program query language, which shares a similar spirit with *Epitaxis*. Whereas *Epitaxis* is a hybrid combination of Prolog and LISP, SOUL is a hybrid combination of Prolog and Smalltalk. The prolog based query system of SOUL is designed to work directly with Smalltalk objects, so to use the system with

another language such as Java some sort of interoperability library is needed to make a connection between the elements of the other language and Smalltalk. In the case of Java, JavaConnect is used as a library to allow a Smalltalk application to reference any Java object, via a proxy in Smalltalk. It is not clear how available this technique is for other languages, especially ones that are not object-oriented such as C. *Epitaxis* creates the connection more directly. In *Epitaxis* there is a parser generator that can be used to create an AST using CLOS objects, so once you have the grammar for the parser, the AST can be built in a form that the query language can work with directly. SOUL has an *open* [22, 88] unification system. This means that some of the inner workings of the unification algorithm are user programmable. This allows for user definable additions to unification, which is used to create a level of abstraction in how objects are declared equal with respect to unification. In *Epitaxis* the ability of more abstract or complex matching is done within the systems' rules (or by direct modification of the unification code base). This extra complexity of matching can be packaged up into functions also creating abstractions. In the case of SOUL the complexity is pushed into the unification algorithm, in *Epitaxis*, the complexity is packaged up into the rule system. *Epitaxis* has a rule-based interpreter to enable queries on running code. This system is *open* and allows user level modification by

overriding and/or augmenting sub-systems within the interpreter in a manner similar to how SOUL allows the user to augment the unification algorithm. SOUL gets the open interpreter for free by using Smalltalk's meta-object protocol, but it is not clear that this functionality will carry over with other languages.

The ram based AST systems allowed for fast processing but ran up against limits on the size of programs they can query. As relational database management systems (RDBMS) became more efficient it became practical to transform the program into a collection of entity-relation (ER) tuples [34]. While this slows down the access it allows for enough memory to handle any size program. One of the earlier systems to use this approach is OMEGA [104]. Another is C Information Abstraction (CIA) [36].

JQuery [50, 83] is another system that uses a RDBMS. JQuery uses TyRuBa [49], which was originally used to create a Parametric Type System for Java. JQuery is a Java browser implemented as an Eclipse plug-in. JQuery is generic and can be configured by writing TyRuBa include files for rendering many different types of views. JQuery passes queries to TyRuBa, which executes them over its source model, a database containing facts about the browsed program's

structure. The results of the query are passed back to JQuery, which creates navigatable views of them.

Java Tools Language (JTL) [39] is a language for querying Java. It is a declarative language based on logic programming which uses a simply typed relational database for program representation. JTL is designed as a “tool for making tools”. JTL is currently intimately related to Java since it queries the binary representation within the Java environment. Because of its program representation JTL is limited in its ability to query dynamic control flow questions. It has a set of *ad hoc* built in predicates to extract this information in a Java environment dependant way. Given these limitations, JTL still has a fairly expressive query language. Its underlying semantics is first order predicate logic augmented with transitive closure.

There are a number of program query systems based on Datalog [31]. Datalog is a database query system based on the logic-programming paradigm. Syntactically Datalog is a subset of Prolog, however this subset differs semantically. Datalog semantics are purely declarative, whereas in Prolog there is an operational semantic (e.g. the order of clauses can effect whether or not a Prolog queries terminates whereas all Datalog queries terminate). Datalog

queries are translated into relational algebra (RA). Each clause of a Datalog program is translated into an inclusion relationship of RA. Datalog is at least as powerful as positive relational algebra but not as powerful as full relational algebra as there is no Datalog rule defining set difference. These expressions can be captured by extended versions of Datalog that use logical negation. Datalog is full of clever optimizations and has numerous implementations.

CodeQuest [74, 75] is a source code querying system based on Datalog. It has two main components. The first is Datalog and the second is an Eclipse plug-in responsible for parsing Java. The use of a disk based RDBMS makes CodeQuest scalable to large-size software systems. The use of an Eclipse plug-in leverages its automatic system for recompiling appropriate modules as source code changes. This way the entire project does not need to be processed to reconstitute the database after a small source change. Datalog queries are translated into a version of SQL that handles recursion, then passed to the RDBMS.

Another system, called Program Query Language (PQL) [99, 107], allows queries to be made on an executing program. It uses static analysis to minimize the number of instrumentation points needed. The queries can also be fixed with

actions to be performed when the query matches. This way potential errors or security flaws can be handled “on the fly” as they happen. The query language is designed to match sequences of events. Lower level actions such as variable accesses are abstracted away. The events tracked are field accesses, array accesses, method calls and returns, object creations and the end of program (to know that something has not happened). PQL essentially looks to find a set of heap objects to parameterize a context sensitive pattern of events of a program execution trace. The system performs flow-insensitive points-to analysis. The points-to information is stored in bddbddb, a Datalog deductive database implemented using binary decisions diagrams. Queries can be nested and can be recursive. A key feature of PQL is its ability to do object-based parametric matching across time in a running instrumented program. Currently PQL works only on Java.

2.1.1 Summary of Systems

There are three components to source code analysis [16]. They are the transformation, the representation, and the actual analysis. Software in its most usual form is a text file organized as a flow of instructions. Because of the multiplicity of elements, representations, relationships, and layers of abstraction

in software most of the information content is implicit. Because it is temporal, information may not become obvious until all the predecessorary actions have been executed and successor states are extant. It is therefore expedient to extract into an explicit form the information of interest before attempting to analyze it. The transformation usually involves some form of parsing. There is a wide range of structures that the source code may be parsed into. These include the abstract syntax tree, the control flow graph, the call graph, the value dependence graph, single static assignment form, ER tuples, the trace flow graph, etc. Often one transformation is used as an intermediate step to a further transformation, as it may be too complex to go directly into the final form. Finally, some form of analysis is performed on the transformation. This may be done to prove some fact to validate an optimization, or simply to extract a highly embedded piece of information. In the case of compiler optimizations the analysis is formalized and scripted towards a specific goal. In the case of query a more interactive generic system is used to try to maximize the exposure of information.

The systems described above show a progressive range of software representations. At the simplest is simply the source code text. The next level is a list of lexical tokens. The next level is an AST. Some systems allow various

annotations to the AST in essence constructing an ASG. Some systems instead of keeping a memory based AST, decompose the AST into ER tuples into a data base. These tuples can also contain the equivalent of AST annotations. Some of the systems are fixed with respect to the AST they build and some allow the user to add arbitrary annotations and edges. These systems also range in the degree to which they can handle non-syntactical elements such as comments, white space, line position, macros, and conditional compilation directives.

There is also a progression of analysis/search techniques. These range from regular expressions to path walking, to path walking with wild cards and variables to full first order logic. With this is a corresponding variance in the ability over the order of search. Of course lexical search only has one search path, however searching through tree requires a search methodology such as preorder, postorder, depth first, breath first, or some form of general arbitrary control.

The literature shows a progression from text based to memory based AST to disk based RDBMS. Most of the systems (except PQL) cannot handle queries over control flow graphs. Most of the examples of queries in the literature are relatively simple. Some of the systems are software engineering systems. That is they handle higher-level aspects such as domain knowledge and software

architecture. *Epitaxis* specifically aims at the act of programming. It is designed to have an extremely fast *query-modify cycle* and to address queries at all levels of code, including not just syntax or control flow queries, but also awareness of runtime values and states. It is designed to push software query into the realm of program analysis.

2.1.2 ER-tuple representation versus AST representation

Tuple representations of class hierarchies, methods, member elements are easy and natural. They are essentially sets. So all the systems above which use RDBMS to represent programs are very good at this sort of query (e.g. “What methods does this class have?” “Is there a methods that returns this type?”, etc.) They easily “capture” set-membership based non-local aspects. These systems have difficulty when representing the imperative (or local) aspects of programs. Imperative aspects cannot easily be represented by sets. You would need an enormous number of follows relations, and other relations, which describe all the complexities of control flow. This is best represented by a CFG. *Epitaxis* has the expressive power to query the AST and derive the CFG, which can then be used for further queries. The RDBMS systems, which get around this problem, do so by partial *ad hoc* measures. PQL answers control flow questions using dynamic

analysis. They instrument the code and answer queries on the trace. This has all the normal pros and cons of dynamic analysis versus static analysis, but it is nonetheless a limitation. Some amount of static analysis is done, but this is implemented internally and not expressible using their query language. JTL slightly gets around this problem by “hard-coding” some primitive queries which analyze the Java byte code to answer dynamic program questions. These types of queries are not generally representable in their query language and so their language is limited by what they have prebuilt. These queries pertain to accessing, reading, writing, and calling, but are unordered within the method body and are therefore flow insensitive.

The query languages which use an AST or graph based representations also have limitations. In ASTLOG the AST is imported and queries are made on it. There is no way within the query language itself to transcend this structure. Given a semantically complete AST representation, the query rules can deduce control flow information, but this can become quite cumbersome and inefficient. It is also a further step away to actually deduce runtime state to perform any kind of dynamic analysis. CodeSurfer gets around this by building a very extensive graph to represent the program. This is done internally and not

expressible in their query language. In fact, their query language is actually not deductive, but simply walks the graph representation. It relies on the information already being represented on the precomputed graph.

A key insight of *Epitaxis* is that representation matters. *Epitaxis* not only has a source complete [103] representation of the program, but also has a powerful enough query language to extract other abstracted representations as needed to allow search on a higher level of abstraction. Also, the expressiveness of its query language and its database of a source complete AST allows *Epitaxis* to be more easily extendable to other programming languages. It does not rely on client language features such as its byte code representation or features of the client language's environment such as reflection to access things. *Epitaxis* only needs a grammar for the language and class definitions for the nodes of the AST; its query language is powerful enough to do the rest.

The important question for *Epitaxis* relative to the RDBMS based systems is how well does it scale to large programs. Speed wise it scales very well as it has a memory based representation. The question that remains is how large of a program can its representation fit within the limits of RAM? The one thing in *Epitaxis*' favor is time. Systems with 4Gb of RAM are common, and with the

advent of 64 bit systems, this will continue to grow. Memory size grows exponentially with time, whereas, program code size probably grows linearly with time. I suspect we are very close to the transition where memory size overtakes program size. I currently estimate that *Epitaxis* can represent 1 million lines of C code in just over 3 GB of RAM.

2.2 Symbolic Execution

Validating software is a complex and difficult task. There are two approaches to ensure the correctness of software. One approach is *program proving* using techniques from static analysis and model checking. While theoretically this approach is very appealing, powerful and accurate, practically this approach does not scale well to realistic size programs, tends to be restricted to simple properties, and is prone to a large number of false positives [144]. The other approach is *program testing*. Testing is expensive, tedious and tends not to find most errors. To remedy this there has been research toward automating the creation of tests. The main method for this is through symbolic execution.

Symbolic execution is a technique to determine a set of input vectors that when input to a program will cause all of its paths to be executed. The most basic

approach to symbolic execution is to first create a control flow graph (CFG) from the program source code. This identifies all the decision points in the program and all the variables which control those decisions. This CFG is then traversed from the entry point along a particular path. A list of all the input variables, variable assignments and branch predicates are then collected. The input variables represent *symbolic values*. The variable assignment expressions and conditional predicates (constraints) along the path represent the *path conditions*. The values of the output variables will then be represented by expressions in terms of the symbolic input values, assignment expressions, and conditional predicate constraints. This is then solved for the input values. If there is no solution then the path is *infeasible* and control flow cannot reach that point. If any solution is found, the path is *feasible*; the values of the input variables determine a vector of test input values to cover that path in the code. The system will then negate the last conditional (to follow the other path), follow that path then solve the new path condition to generate an input vector to reach this new end point. The system will continue to backtrack, negating the new last conditional, and continuing until all the paths are covered. There are now a set of test input vectors which can be used to exercise the code base. Mathematically, symbolic

execution divides the input space of a program into a set of equivalence classes, each defined by a path through the program.

The goal of the approach is to generate a set of input vectors that will cover all the execution paths through the program. There are two serious limits to the above technique. One is path explosion and the other is that solving the constraint equations may be extremely difficult. In practice, this both limits the coverage produced and covers many infeasible paths. Often the same region of code can be reached via many different paths. In order to help reduce the number of paths followed, the systems keep track of where they have been and stops the system from producing redundant coverage.

One of the earliest systems using symbolic execution is EFFIGY [92]. It worked on a "simple" version of PL/1. Another is SELECT [20] working on a subset of LISP, and one by Clarke [37] working on ANSI Fortran. All of these systems were very limited research systems. In addition to the limited processing power of their day, these systems were also limited by their ability to only handle primitive data types, e.g. integer, Boolean. An increase in practicality and power came with the ability to handle dynamically allocated data structures (e.g. lists and trees) allowing use on modern programming languages such as Java and C++ [86]. This approach is called *generalized symbolic execution* by Kurshid. In

addition to generalizing the system to handle dynamically allocated data, their system also used a source to source translation to instrument the code. This allowed their system to perform symbolic execution using a standard model checker (for the underlying language) instead of having to build a dedicated tool. It uses *lazy initialization* to defer the actual allocation of heap objects until a field in the object is first referenced. This way only the portion of the heap that is referenced along the current execution path is materialized. There are three possibilities in initializing the referenced field: 1) a null value, 2) a new object of the field's type, and 3) an existing object of the field's type. This creates 2 or more branches in the execution tree. The third case can create 0 or more branches depending on the number objects of the field's type the path had already created. However, their system was still not practical for real world applications.

2.2.1 Concolic Execution

Another variant to symbolic execution is *concolic execution*. Concolic execution is a combination of *conc*rete and symb*olic* execution. By instrumenting the source code with all the functions needed to carry out the symbolic execution, the program itself ran the symbolic execution along with a concrete run. This avoided a whole level of separate machinery to do the symbolic execution. The system also had concrete values available when the constraint

solver got stuck. This allowed symbolic execution systems to reach a real level of practicality. Part of the efficiency of this method is that the test runs are happening simultaneously with the symbolic execution, not in a separate run after the symbolic execution has determined the input values. Here if the concrete execution throws an exception a bug has been found. Of course the exception may occur arbitrarily far after the cause. Bugs can also be missed if they are only caused by a subset of the values in the equivalence class represented by the path. As an example if there is a condition `if (abs(x) < 10) then a = y / x;` In this case `x` having concrete value of 3 is in the equivalence class of values that may have gotten execution into this path, but it will not trigger the error.

DART (Directed Automated Random Testing) [71] uses this approach. DART executes the instrumented program repeatedly, the first time with a set of random input values. If the input variable is a pointer DART will randomly initialize it to NULL (with a 0.5 probability) or with the address of a newly allocated memory location, whose value is in turn initialized using the same rule recursively. Constraints on pointer data are thereby avoided in favor the randomly generated data. Each additional run uses a record of the conditional statements executed in the previous run. The conjuncts in the path conditions are

systematically negated. This is used to generate the next set of input values needed to follow the next path. Thus DART attempts to follow all feasible paths. If the constraint solver is unable to solve a conditional the symbolic condition is replaced by the concrete value, then both the concrete and symbolic execution resume with it. By picking the branch determined by the concrete value (derived from the initial random values) instead of choosing both (which may result in unsound behavior if the other path is infeasible), bugs found along the path are sound. A traditional symbolic execution system which cannot solve a constraint will not know how to generate a value to take either path and will either stop, possibly missing a feasible path (producing incomplete coverage) or choose both, possibly taking an infeasible path (possibly producing an unsound result). The difficulty in the DART approach is to provide the methods which extract and solve the constraints generated by the program [124].

Another system using *concolic* execution is CUTE (Concolic Unit Testing Engine) [124]. CUTE allows constraints on pointer based structures by separating pointer based constraints from integer constraints and simplifying them. This allows them to represent and solve approximate pointer constraints to generate test inputs. CUTE represents pointer input values *logically*. That is they don't have physical memory addresses. When a pointer input value is referenced

CUTE first explores the path where the value is NULL. When this constraint is negated, CUTE must make the pointer point to a structure of the appropriate type. To populate this structure CUTE will randomly generate numbers for numeric fields and follow the NULL and non-NULL logical pointer paths when the inner pointer fields are referenced. These pointer values are represented in a *logical input map;* they are not part of the physical address system. Constraints in the logical addresses are represented by constraints on the integers that represent them. Each logical address also has a type associated with it. The logical address system of CUTE is a much simplified version of the *proxy-value* system that *Epitaxis* uses.

One of the main problems with both the symbolic and concolic execution systems described above is that due to the possibly enormous number of paths that must be explored symbolically, the testing tends to be *wide* but not *deep*. That is they explore many of the paths from the initial starting point, but because there are so many they never explore them very deeply. This problem is particularly prevalent on programs with complicated input or even programs which input a text string. The systems explore the input *parsing* code (mostly exploring paths detecting malformed input as even a text input string of only 10 characters long has 72^{10} possible values (assuming 72 valid text characters), most

of which are probably illegal) but never make it to the input *processing* section of the code. To address this problem *hybrid concolic testing* was developed [106]. This system interleaves random testing with *concolic* execution to explore the program state space both *deep* and *wide*. It uses CUTE to do *concolic* execution. *Hybrid concolic testing* starts by testing random paths. This allows the algorithm to explore deeply quickly. When the random testing *saturates* (finds all the paths with large equivalence classes and does not produce any new coverage after a set number of steps) the algorithm switches to *concolic* mode (from the current set of program states) to find the improbable paths (those with small equivalence classes that take a precise set of input values to find) thereby achieving *wide* coverage from a set of *deep* points. *Hybrid concolic testing* is most suitable for testing programs that periodically get input (*reactive* programs) and not suitable for programs which get fixed initial input and then process the data (*transformational* programs). Figure 1 shows their motivating example. It is an abstraction of a state machine taking both a character and string input iteratively within an infinite loop. The problem is to cover the execution space to find the ERROR at the bottom. The authors state that both pure random testing and pure *concolic* testing (presumably using CUTE) were unable to hit the ERROR after

one day of testing while their hybrid system hit the ERROR in a couple of minutes.

```
void testme()
{
    char *s;
    char  c;
    int    state = 0;

    while (1)
    {
        c = input();
        s = input();

       /* a simple state machine */
       if (c == '[' && state == 0) state = 1;
       if (c == '(' && state == 1) state = 2;
       if (c == '{' && state == 2) state = 3;
       if (c == '~' && state == 3) state = 4;
       if (c == 'a' && state == 4) state = 5;
       if (c == 'x' && state == 5) state = 6;
       if (c == '}' && state == 6) state = 7;
       if (c == ')' && state == 7) state = 8;
       if (c == ']' && state == 8) state = 9;

       if (s[0] == 'r' && s[1] == 'e' && s[2] == 's' && s[3] == 'e' && s[4] == 't' && state == 9)
          ERROR;
    }
}
```

Figure 1: A Simple Test Function

The *concolic* execution systems described above use a depth first exploration (DFS) strategy by repeatedly exploring a new depth with each iteration. EXE (EXecution generated Executions) [29] works using DFS by forking at each decision point instead of keeping information in an external file and rerunning the program. This way the number of children is linear in the depth of

the process chain. EXE works by using a source to source translation to instrument the program with the routines necessary for the symbolic execution. In addition the user must manually insert calls to mark areas of memory as symbolic. This presumably limits the scope of what is executed symbolically so the program can run without overflowing memory tracking. Whereas CUTE and DART tightly interleave symbolic and concrete execution, EXE merges them. If the operands are concrete, normal concrete operations are performed. If any operands are symbolic then symbolic operations are performed. EXE uses a UNIX fork() system call at each unresolved decision point. When taking the true path it assumes the condition true and when taking the false path it assumes the condition false. The fork() points constitute a DFS chain of the execution tree. In this manner EXE (ideally) will follow all paths. EXE's instrumentation includes *universal checks* for integer division and modulus by zero, that a dereferenced pointer is never NULL, and that a dereferenced pointer lies within a valid object. In addition programmer supplied assertions are also turned into a *universal check*. A *universal check* is more general than a concrete check. The *universal check* tests that the error condition or assert condition are not possible for any possible solution to the symbolic value tested. So whereas with *concolic* execution a program fault will occur only if the concrete value standing in for the symbolic

value triggers the fault, universal checks will detect the error if any possible feasible value triggers the fault. This way, one path traversal represents the whole equivalence class of values, not just one concrete vector of values in its error detecting ability. Of course this error detecting is limited to the small set of universal checks that are instrumented into the code. EXE will create a concrete value(s) to continue execution if its constraint solver fails.

KLEE [28] is a system that works using a custom built virtual machine. The KLEE virtual machine directly interprets the assembly language output by the LLVM compiler [100]. The virtual machine operators handle both concrete values and symbolic values. Potentially dangerous operators (such as division or pointer dereference) generate branches that check if any input values could cause an error. If one is detected, KLEE generates a test case and terminates the execution state. On conditional branches, KLEE queries the constraint solver to determine if the conditional is either provably true or false. If not KLEE clones the state so that both paths are explored. When a dereferenced pointer can refer to N objects, KLEE clones the current state N times. KLEE has been designed to handle the two following problems with symbolic execution: 1) handling the exponential number of paths through code and 2) the challenges in handling code that interacts with its surrounding environment, such as the operating

system, the network, or the user (colloquially referred to as "the environment problem"). KLEE is a robust system that has been tested on a wide range of real programs. During its testing it reached a maximum of 95,982 concurrent execution states. The average of the maximums across programs tested was 51,385. KLEE implements various state compaction optimizations as well as query optimizations to achieve these results. Like *Epitaxis*, KLEE maintains the entire execution tree and uses a memory management system that shares values across states. KLEE models about 40 systems calls (e.g. `open`, `read`, `write`, `lseek`, etc). This allows it to write symbolic values to a file and later read them back in. KLEE can also be set to simulate environmental failures by failing systems calls in a controlled manner.

KLEE gets more power by virtue of implementing their own virtual machine to directly handle the symbolic execution instead of having to instrument the source code. *Epitaxis* takes this further and has its own "source" level interpreter. This allows access to the source level constructs, declarations, and type definitions. Much of this information is compiled away and is not available to an assembly level virtual machine. *Epitaxis* uses this information to know what types are being referenced, when members are being referenced, and

about casting. This allows a much wider range of information to be queried or checked.

There are two main thrusts to the work of symbolic execution. The first is effective execution tree coverage. This means that the SE system should traverse all the feasible execution paths (completeness) efficiently and without redundancy, and without traversing any infeasible paths (causing unsound results). The second is either producing the input vectors to test programs or directly testing the code during the coverage. Currently the types of bugs found are "crash" bugs. (The one exception to this is the systems which track object size and catch when pointers are dereferenced to locations outside the defined boundary). These are bugs that are "easy to spot" once you are at their point of execution in the running code. The CPU itself will point them out to you (by throwing an exception). Although it is good to know about these really what is wanted is not this symptom, but the cause i.e. where the code goes wrong in the first place.

Because of the typically exponential growth of execution paths and loops whose termination depend on symbolic values, symbolic execution systems have to enforce alternative termination strategies. They cannot wait for all their paths to reach an exit or error state; some never will. To ensure termination within a

reasonable amount of time they either impose a depth limit to the execution tree or a run timeout limit. *Epitaxis* adds a third alternative to these; it can track how many times a path steps on itself. After a path traverses the body of a loop and tests the loop exit condition, assuming a symbolic value, it will split into two. One leaves the loop and the other repeats the loop stepping on itself. *Epitaxis* can be set to allow this to happen a set number of times after which the path will terminate. This is useful if the system is looking for type anomalies in assignments. Except in contrived cases these are likely to be found the first time through the loop and a second iteration is unnecessary.

This research aims to expand this second thrust in two ways. First is to spot these errors sooner, that is closer to their cause and the second is to use this powerful execution space coverage technique to find more "sophisticated" and "conceptual" bugs. For this prototype version this involves both the interpreter tracking more semantic level information that can be used to locate bugs on the spot and gathering information from throughout the execution tree about type usage and member access and analyzing it for problems as a collection. We expect that further research can expand this class of analysis. Part of the reason for the *open* design of the symbolic execution is to facilitate exactly this line of research.

Additionally, this research leverages symbolic execution's ability to traverse the execution space in order to bring program query into the semantic realm. Because of the *open* architecture of the symbolic interpreter it is possible to attach assertions and collectors to semantic points in the execution space. This will be described in Section 4.4.7 below.

2.3 Aspect Oriented Programming

Aspect oriented programming (AOP) is a kind of meta-programming system for addressing *crosscutting concerns* [130]. *Crosscutting concerns* are parts of a software system that would logically belong to one module (called an *aspect*) but implementation-wise needs to be distributed throughout the software. AOP and *Epitaxis* share a common concern of how to label and find (or select) a related set of points, called *join points*, within software. In the case of AOP these *join points* specify where pieces of code need to go. Some AOP systems can recognize both static and dynamic *join points*. A *static* join point can be defined by a location in the programs source code or syntax tree without need of runtime information. A *dynamic* join point in addition specifies runtime information such as memory state or call sequences. AOP languages which allow static *join points* are called *specification based* join point languages. One which also allows memory state to

specify locations are called *state-based* join point languages, and one which also allow call stack information are called *program based* join point languages. Whereas AOP languages specify *join points* to define where to effectively insert code *Epitaxis* specifies *collect points* to locate where to extract information.

2.4 Object-oriented Logic Programming Systems

A related area of research is in object-oriented logic programming systems. Although this is not a main thrust of the current research, an object-oriented logic programming language forms the backbone of the analysis engine. This system has some interesting features related to those, which show up in the literature. Object-oriented logic programming systems are prolific in the literature. An annotated bibliography from 1993 [5] states that there are about 180 references and at least 50 different mergers and/or object-oriented plus logic programming languages. They can be divided into three groups based on 1) an object-oriented language with logic added, 2) a logic language with objects and/or inheritance added on, and 3) languages designed from the ground up to have a somewhat equal expression of both. The system described in this thesis is in the second category. Typically these systems use object-oriented ideas to implement some form of inheritance or as a way to create different name spaces

for rules and facts to help organize the database. The other synthesis is to augment the structure of facts. Logic languages often have a "purest" [30] quality where the only data structure they admit is lists. By incorporating facts as objects the facts can have a little more structure. This can go to the point of full frame based systems or semantic nets with logic programming on them. *Epitaxis* uses the class structure of facts to help organize its database of rules and their range of applicability. *Epitaxis* also uses the object structure to organize its database of facts, however because this database is generated by a parser or by rule application, it tends to have a much more regular structure than a frame based or semantic net system.

"If you don't know where you are going, you will wind up somewhere else."
--Yogi Berra

Chapter 3

3. Background

Software is an entity that has meaning on different levels: 1) character; 2) lexical; 3) syntactic; and 4) semantic. Information must be lifted and integrated through these levels in order to gain an understanding of what a program does. A software developer must gain this knowledge and understanding in order to locate problems and make changes to a software system, so that the system can stay current with user and business needs. This process relies heavily on human reasoning to analyze and integrate information, but as software increases in

quantity, size, and complexity, tools that are capable of taking over more of the work are needed.

In order to support this process of information integration, an environment is needed where software developers can interact with information about source code from the four levels. Each of the different levels of meaning and abstraction for source code has their uses, and different levels are inherently more efficient for different types of queries or analysis. *Epitaxis* forms the basis for such an environment. This approach builds on research in five areas: source code searching; parsing and interpreting; unification and deductive retrieval; program query; and symbolic execution [76, 99, 108, 109, 117, 122, 123, 132, 134, 135, 141]. Search is a technique to identify things; it is a way to name an unknown entity such that it can be found and processed. The more expressive the search language, the wider the range of things that can be identified, and consequently the more that can be dealt with computationally. Parsing is essentially a technique to transform representations. Different representations determine how easy it is to access content. Unification and deductive retrieval is a way to reason over some representation. Symbolic execution is used as a technique to expose the execution state space to search. Finally, program query

can be thought of as search on a transformed representation which permits search at a higher level of abstraction, that is, for function and relationships instead of a search on parts. Much of the labor in program analysis is the construction of a representation that admits the desired results and must be tailored for each new analysis. This is a key insight of this research: a robust and flexible unified representation allows analyses to be performed using a declarative and constructive query language to identify a variety of results, ranging from character-based search to value-based search, including syntactic and semantic query.

3.1 Source Code Searching

Software developers have needed tools to search through source code since the beginning of interactive programming environments. It started with simple keyword search. This was extremely limited as it only allowed you to find fixed sequences of characters without regard to context. When regular expressions were added, it became possible to find search patterns and context using the search language. An important advance was made when search techniques started using concepts of program structure, such as identifiers of variables and functions, directly in expressing search patterns. Syntactic search is

more difficult than lexical, due to language specificity and implementation challenges. There is more possible variation in the order and direction of search in a tree structure than a linked list. Syntactic search is usually limited in its ability to reason about its choice of search direction. This is largely a limitation caused by the expressiveness of search languages. Often search direction is fixed such as in preorder traversal, or by only being able direct movement based on immediate neighbors. One example of such a search language is syntactic regular expressions [4, 12, 119, 120]. They are limited by the limited expressive power of regular expressions. In addition, they have no dynamic representation. The usual solution is to build another data structure in which the search direction decisions are easier to make.

Another approach to search at the syntactic level involves pre-processing the program and storing facts in a database file of entity-relations [34]. The CIA System [36] uses this approach, as do others [6, 32, 75, 104]. Alternatively, the parse tree can be transformed into other representations such as data flow graphs or control flow graphs [7, 59, 77, 97, 105, 116].

The various methods above each address part of the problem. Because they are based on a fixed data structure or a limited search language they are

each limited in what semantic information they can access. Also, the set of relations is fixed at build time so if the needed relationship was not anticipated in advance there is no way to search based on it. These analyses are run in batch mode on a snapshot of the code, which limits their applicability in today's integrated development environments.

Searching at the semantic level is a problem in need of original research. While information from semantic analysis can be stored in a database and queried in the same manner as syntactic information, this approach is limited to those relations and keywords that have been stored. There is no general mechanism for specifying patterns and relations at the semantic level.

This research seeks to provide a mechanism for specifying arbitrary queries and relations in source at all four levels of program structure. This mechanism is based on Prolog-like predicates, consisting of facts and rules that can be used to specify software entities including individual characters, classes, dependencies, abstractions, and even execution patterns or relationships. The desired search target is represented as predicates that the *Epitaxis* system uses to traverse, i.e. search, a unified data structure that includes the lexical

representation, an abstract syntax tree, a control flow graph, execution states and virtual abstract semantic graphs.

Typically, Prolog systems search a collection of facts and use rules for combining or inferring facts [18]. Object-oriented versions of Prolog [66, 82, 115] allow the list of facts to be grouped and to inherit from other groups of facts, but their object-orientation exists primarily to create name spaces for groups of facts. Various relationships between parts of facts can be found through unification, but the search path is linear except when rules are followed. The *Epitaxis* approach differs in that the facts are stored as a tree or graph of objects or can exist virtually, as well as being formed on the fly, rather than as a simple unstructured collection of facts and rules. Whereas Prolog finds *logical* relationships between facts, *Epitaxis* finds *structural* and semantic relationships between objects and execution states. Consequently, the search paths and targets can be more contextually related, more creative, and much more expressive.

3.2 Parsing

Parsing came about as a means to bridge the gap between how humans interact with source code and how machines interact with source code. Parsing

takes a sequence of tokens and converts it into a tree, such that the software's syntactic structure is directly represented. This process is so important that early in the history of electronic computation, it was studied and vastly improved; parsers became much more efficient in both time and space [52-54, 95]. Parser generators were created that allowed a parser to be built from a language specification [2, 33, 93].

Once parsing became commonplace, ASTs became a standard data platform for further transformation of representation. Like lists of characters, or lexemes, trees of syntactic entities are limited in what relationships they can directly express. (However, they are good at expressing context or scope.) Indicative of the chasm between syntactic and semantic structure, parsers and parser generators are unable to go directly to a general semantic structure. Analyzers are needed to create decorated ASTs, or abstract semantic graphs [15]; these, or equivalent data structures, were created to directly expose more complex relationships within software such as data flow, control flow, and dominance [21, 45, 46, 102].

Like the tip of an iceberg, this diversity of data structures and techniques hints at an underlying difficulty. The problem is that these techniques are trying

to squeeze out semantic information from a static structure. This works well enough in individual cases, but precludes easy integration of these analyses. Every algorithm uses a different static structure to capture the behavioral information that it needs.

The gap between syntactic and semantic representation is not just quantitatively greater than the gap between lexical and syntactic representation but is also qualitatively different. Semantic relationships are not just spatial but also temporal. They are not fixed; they depend not only on *where* in the code, but also on *when* in the code. In some cases, an abstract summary of information is needed to enable the system to be efficient. This means that any fixed data structure can capture only a small subset of the content.

The solution to bridge this gap between syntactic and semantic representations lies not in a fixed data structure, but in a dynamic representation, specifically, the virtual abstract semantic graphs. Rather than constructing abstract semantic graphs as needed by each analysis, the *Epitaxis* approach declaratively reifies the required ASG while traversing a representation of the source code to provide the equivalent information. This approach has the advantage of being able to represent an arbitrary set of abstract semantic graphs

simultaneously in a single data structure. The set of graphs is limited only by traversal rules stored in the system and this collection of patterns, templates, and functions is completely extensible.

3.3 Unification and Deductive Retrieval

First-order logic and logic programming is used across a wide range of domains. One of the first suggestions of its use in computers dates back to McCarthy [110] in 1959 for representation and reasoning in AI. Logic programming also had an early start in mathematical theorem proving [68]. It eventually found its way into a general-purpose standalone programming language know as Prolog [51]. Prolog and its descendants have evolved and there are now many systems with non-standard versions of logic, including modal logic, temporal logic, many valued logic, default and non-monotonic logic, fuzzy logic, etc. It has also been used in areas of program analysis such as model checking. One of its benefits stems from the fact that it is declarative instead of procedural. This makes writing Prolog like programs easier (at least once you understand the paradigm). A drawback of Prolog type programming is efficiency, completeness and termination issues. Some of this has been mitigated using tabled or memoized systems [35, 133].

Typically, Prolog systems operate with an unstructured database of facts and rules. In practice, there is a simple ordering structure imposed on the database by the order in which the facts were entered and consequently, the order in which the facts are searched. The facts represent what is known and the rules express what can in addition be logically deduced. Prolog systems use backward chaining to deduce if some assertion is true or not. It also uses unification to bind the facts and terms that will make some assertion true. In this way it performs a search. One of the difficulties of Prolog systems is that they are "timeless" and "contextless". Like its Platonic ideal (logic), Prolog tends to see truth independent of time or context. Since many man-made systems don't meet this ideal, Prolog variants have been created, such as, object oriented versions [14, 65, 89, 90, 115], and non-monotonic versions [69] to help organize context or time dependence.

In this research, an alternate form of unification and deductive retrieval is explored. Instead of operating on a "flat" set of facts the *Epitaxis* system operates on structured data. For program query and analysis this includes the AST and CFG of the program. Here each *fact* is a node in a tree or graph. The node's class corresponds to the *predicate indicator* and the values in the slots of the node

correspond the *terms*. The slot names supply the term ordering. In this system, unification binds variables to positions in the tree or graph, and rules express movement through the tree or graph. In addition, rules can mandate the creation of new nodes, which can have their slots filled through unification. With the additional feature of *unification by reference*, where variables can bind to locations within objects, the system can, by using rules to walk structures, build *epitaxic* structures linked to the original structure bi-directionally. This mechanism is used to create a CFG from the AST. In addition, the current research adds two features to address the lack of context sensitivity in Prolog. The first is a predicate that declares dynamically scoped variables. The variables in a Prolog rule are statically scoped. By introducing a predicate that declares some variables *special* they can capture the dynamic context that is expressed within many AST structures[1]. For example, the semantics of an AST node for a while loop dictates the scope of break and continue statements. Any rule fired within the body of a rule processing while statements will want a dynamically declared target for any processed break and continue statements. The second feature to address context is that the rules are "methodized". Each rule is specialized based on the class of

[1] A similar idea has been incorporated into the LALR(1) parser productions to create dynamic symbol tables to handle the dynamic and messy way that C identifiers can be types or variables and shadow each other.

the first term in the head predicate. The signature can be generalized to honor the class of all the terms, but this extra feature has not been needed for the current application. Other non-standard mechanisms are used to structurally memoize nodes as they bind to variables within certain contexts. *Gatherers* collect the nodes the variable binds to and *generators* emit the nodes that have been collected. The final enhancement to the system is the ability of the search routine to call upon the symbolic interpreter to present execution states to be searched. The idea here is that the system can "deduce" or find things not by just searching what already exists in the database (the AST), but can also find things in structures that it builds by virtue of searching other structures, or by searching states in an executing program.

3.4 Program Queries and Analysis

There is a spectrum of program analysis techniques, including control flow analysis, data flow analysis [78, 79, 114], program dependence graphs, slicing [64, 80, 125, 136, 138, 139], pointer alias analysis [47, 81, 113, 128, 129], type inference [19, 73], abstract interpretation [40, 85], symbolic execution [11, 38, 42, 43, 91, 92, 94] among others. They are often broadly divided into two classes: static analysis and dynamic analysis [9, 60], although other taxonomies have

been suggested [145]. Static analysis techniques work on some fixed data structure representing some aspect of the software [27, 56, 61-63]. Often constructing this data structure is the bulk of the work, because it determines the kinds of analyses that can be done easily. In some sense static analysis builds a model of program state, which is an abstraction of the runtime states. The system then reasons over the abstracted model. The nature of the abstraction determines the type of static analysis performed and its properties. By being willing to throw away information (creating the abstraction) the analysis trades precision for soundness. The result is more accuracy (soundness) on a simpler (conservative) problem. Static analysis can be said to focus on a subset of data structures. Dynamic analysis is used to provide information about the program behavior at run time. There are a variety of approaches, including producing an instrumented version of the executable, and augmenting virtual machines [10, 13, 24, 112]. In dynamic analysis the program is executed and facts are collected. Here the information is completely precise, but specific to the particular run (the set of inputs). The results will not usually generalize. Here the analysis trades for precision at the expense of generality (incomplete). The quality of the input and number of different runs the data is gathered over determines the degree of

generality of the conclusions. Dynamic analysis can be said to focus on a subset of executions.

Program analysis systems tend to be procedurally based, monolithic, and complex. Typically, they are designed to be part of a compiler, used to determine the appropriateness of various optimizations. On the other end of the spectrum, there are program query systems. These systems tend to be more declaratively based and interactive. A detailed description of some program query systems is given in Section 2.1. They tend to transform the program to be queried into an AST, a set of entity-relations in a database, or some form of crystallized execution trace and lookup information in the transformed version.

There are a number of features that characterize the expressiveness of program query languages: *population*, *dimensionality*, *direction*, *grip*, *reach*, *container*, *points of origin*, and *language*. 1) *Population* is an expression of the entities that make up the search space. It can be characters, lexemes, pieces of an AST or CFG, some transformed version of these, a set of ER-tuples, or some aspect of an execution trace or execution state. 2) *Dimensionality* refers to the number of orthogonal directions that can be considered when moving through the search space. This describes the structure of the search space, i.e. are we

searching down links of some node, are we searching up or down an inheritance hierarchy, are we searching amongst logical relationships between items, are we searching computed values, are we searching along an execution path, etc. 3) *Direction* refers to the structure and number of different paths in which a search can proceed within a given dimension at a given point. If we were searching through links in nodes there would be one direction for each allowable field. If we were searching inheritance relationships search can go either along sub-classes or super-classes. If we are searching an execution path can it go either forward or backward. If we are searching a logical space, we may only look along true facts, or may also look along may-be-true facts, or even some more general lattice [48] or bilattice [70] of states. 4) *Grip* refers to the quality of what defines a match between the search description and a member of the population. This can be a thing such as the objects identity, value range, class membership, slot values, structural relationships, statistics, etc. 5) Given some reference point or landmark *reach* refers to the range of possible search space between it and the next point of consideration. Some methodologies can only look at the next point along a path, some can also look backward, and others may be able to skip over a large block of the search space to find what they are trying to match. This also refers to the level of control in selecting or avoiding various dimensions or

directions. 6) *Container* refers to the shape or structure of results found. This can be a single item, a set, a stack, a queue, a tree or a graph. 7) *Points of origin* express the variety of possible starting points for the search. Do all searches start at the beginning or root or can a search be started from any point in the search space? 8) The final feature is the nature of the search *language* itself. Is it concise or verbose? Is it procedural or declarative? Is it simple or complicated? Does it admit recursion? Are their looping or other control constructs? In what forms can it hold state or is it stateless? Is it sound? Is it complete? Is it precise? Is it extensible?

From the point of view of information integration to support human reasoning, the overhead cost of many queries and analyses are too high to be run interactively. Typically, each time the code is changed, a new program or executable needs to be analyzed from scratch. A more serious problem is the discontinuity between syntactic and semantic information, because the latter needs to incorporate information about behavior, temporal relations, states, and data values. The two operate on different representations of the source code and the premises underlying the analyses are radically different. There is no mechanism that can be used to specify both types of query. The *Epitaxis*

approach will make use of a data representation and search engine that will allow syntactic and semantic queries to co-occur naturally. Also the data structure representing the program must have a very close structural relationship to the program so the representations are intuitive, are very fast to incrementally update, and allow for high bandwidth interaction. Although one has to contend with learning a very expressive search language, this one framework is capable of expressing queries over lexical, syntactic, semantic, and abstracted views of the program. The current research project intends to push the expressive power of program query near or into the zone of program analysis while maintaining as much of the declarative nature as possible.

3.5 Symbolic Execution

Modern symbolic (and *concolic*) execution systems are an interesting extension of dynamic analysis technology. The systems either instrument a virtual machine (if the language uses one e.g. Java), write an instrumented virtual machine for the assembly produced by the language's compiler or add instrumentation to the source code. The instrumentation then allows the code to execute with symbolic values in addition to the normal concrete execution. The instrumentation collects the path condition, controls backtracking when

exploring alternative paths, and allows the use of concrete values when the constraint solver fails. Framed in this dissertations' terminology symbolic execution systems can be considered as a transformation from the program's static structure (the CFG) to the program's dynamic structure (execution states). Traditional symbolic execution systems search for input values to force execution down specific paths. These values can then be used to drive testing to search for bugs. Newer symbolic execution systems include specific bug tests with the instrumentation and can find bugs such as division by zero, NULL dereferences, and pointer dereferences lying outside a valid object.

In this research, the separate testing phase is completely done away with. Instead of instrumenting the code or the virtual machine, a source level virtual machine is created which detects violations of the language's semantics and the bugs are reported during the search of the control flow graph. What is detectable is not limited by the specific instrumentation.

In addition, this research uses symbolic execution as a means to search the semantic execution space of the program. This represents the execution tree of the program. By building the symbolic interpreter (SI) to work on the ASG directly instead of assembly code, and because the ASG contains syntactic and

type information, the SI has a more complete representation of the execution state at each point in the execution tree. When the SI goes to write a value it is not just pushing a number to an address, but it knows the type of the object being written and it knows the type of the memory location, and even the member name of the place. These can be tested to trigger an assertion or a collector.

3.5.1 Constraint Solving

One of the key elements in a symbolic execution system is the constraint solver (CS). By definition a symbolic execution system has symbolic values. This really means that the system has to be able to compute with unknown or partially determined (constrained) values. This becomes important when a conditional which depends on symbolic values is being executed. This bears on the system behavior in two ways. The first is in deciding which branch of the conditional to take. There are three possibilities and one impossibility. The second is being able to deduce the values of some of the unknowns in the conditional.

The first possibility is when the CS can decide the conditional in spite of unknown values. This is of two kinds: generally solvable and specifically

solvable. An example of the first case is if (x * x >= 0). This is always true whatever the value of x (assuming real x) and represents a poor coding style. A more subtle variation is with nested conditionals where the outer conditional forces values on the inner conditional that then can be resolved only one way. An example of this is if (!x) { if (x * y > 10) }. The second kind is where the solvability depends on the particular run. An example of this is if (x * y >= 0) where during a particular run both x and y are constrained to be positive by prior conditionals but this is not always the case for all possible runs. The system must only follow the "one true path".

The second possibility is when both values of the conditional are possible. In this case there is simply not enough information to decide. The system must follow both paths under their new respective constraints.

The third possibility is that the CS is not powerful enough to decide which value(s) of the conditional is possible. Often constraint solvers only work on linear constraints, quadratic constraints will be unsolvable. The problem here is what to do? There are three choices: 1) take both paths, 2) take neither path, or 3) randomly choose. This can create two problems: 1) taking a path that is really not possible and 2) not taking a path that is possible. In the first case unsound results can follow. The system will think it found a bug that is really not possible. In the

second case results can be incomplete. A path that may contain a problem will not be explored.

The impossibility is when the CS thinks that neither condition is feasible. This represents a bug in the CS.

In addition to deciding conditionals, the CS can also deduce values, or further constrain values for the unknowns. Most systems just collect constraints on the unknowns. So as the symbolic values make their way through code there is an ever growing list of constraints attached to them. Systems often work to simplify this list of constraints or recognize when the constraints have become infeasible. There are also systems where sub-constraints (constraints that are shared) are cached. *Epitaxis* does not explicitly represent constraint lists. Instead it uses a weaker but faster system. The values themselves represent the constraints. There is a symbol `UNSPECIFIED-VALUE` for a completely unconstrained value. If a value is known to be greater than zero (i.e. it is inside a conditional `if (x > 0)` instead of carrying around the constraint (x > 0) the value is represented by an *interval tree* containing only the open interval (0, inf). If later we cross `if (x != 3)` the value is represented by the interval tree with two intervals (0, 3) and (3, inf). *Epitaxis* knows how to do math on a combination of concrete values and *interval-tree* values. The simplification and weakening of the CS

system comes from how UNSPECIFIED-VALUE is handled. If x is UNSPECIFIED-VALUE and y = x + 1 then y will have the value UNSPECIFIED-VALUE. So later if the system executes if (y > x) *Epitaxis* will not know that this has to be true and the system is stuck with the dilemma of choosing between incomplete and unsound when it shouldn't have to. The nature and the power of the CS largely determine the quality of the symbolic execution system.

"You think you know when you can learn, are more sure when you can write, even more when you can teach, but certain when you can program."
--Alan Perlis

"What I cannot build I cannot understand"
--Richard Feynman

Chapter 4

4. Methodology: *Epitaxis*

More powerful program comprehension tools are needed to help software developers understand and maintain increasingly complex legacy systems that businesses and citizens rely on. Current program comprehension tools are hampered by a number of limitations: they lack an appropriate data structure for integrating information about a program from the character, lexical, syntactic, and semantic levels; they lack a mechanism for specifying searches on all these levels of abstraction; they perform syntactic and semantic program queries

separately; they tend to have limited expressiveness of search targets; and finally, they tend to operate in batch mode, separately from interactive development environments.

The solution is to build a software development tool, which can represent software on character, lexical, syntactic, and semantic levels and search through these representations, even interleaving them, to access arbitrary patterns. This will allow software developers to integrate information from the widest range of program queries, including lexical, syntactic, and static and dynamic semantic queries. In addition, the system is embedded within a hybrid character/structure editor enabling high bandwidth interaction and visualization.

The solution has four main parts: a data structure for representing information about the source code at all levels of abstraction, a transformation system to produce these representations and create new ones, a language for specifying searches, and powerful search engine to process them.

The data structure is the Virtual Abstract Semantic Graph. While most software query tools use a fixed data structure to represent semantic information, such as a control flow graph or a data flow graph, this system has a set of

language-specific Prolog-like rules that reify arbitrary ASGs by traversing the source complete AST. This way, as software is modified, and the memorizations become stale, the semantic rules can recompute them whenever the data structure is queried. The semantic rule base will describe how to walk the data structures semantically, thereby allowing the information to be obtained dynamically and interactively. The rules need only describe the relationships between the data elements, the order of execution and the computation and flow of data values.

A powerful search engine (with multi-pathed control flow) can extract any semantic information expressible in its search language by walking the parse tree or the control flow graph using the semantic rules; Semantic patterns can be extracted via unification with variables in search templates. The semantic rules can be tagged with side effects, which will allow the code to be executed concretely, abstractly, and symbolically, or in any mix of the three. These semantic rules with their attached method calls along with a symbolic multi-pathed memory system constitute an *open* symbolic interpreter. All of these components are built into a flexible, extensible interactive editor.

In the remainder of this chapter, these parts, and an initial prototype are described.

4.1 Virtual Abstract Semantic Graphs

In order to make searching source code efficient and robust, the source must be converted from its character-based representation into a representation that more closely maps to its content. Since a program's content exists on four levels, lexical, syntactic, semantic, and via abstraction, the data structure needs to be represented and must integrate information from all these levels. Each representation is structured to map isomorphically to its content. The data structure consists of a doubly linked list to represent information from the character and lexical level that is intertwined with an AST and a CFG. Virtual abstract semantic graphs are reified by traversing this data structure to yield the information needed for queries. In addition the four levels have links between them to allow easy transfer from one level to another.

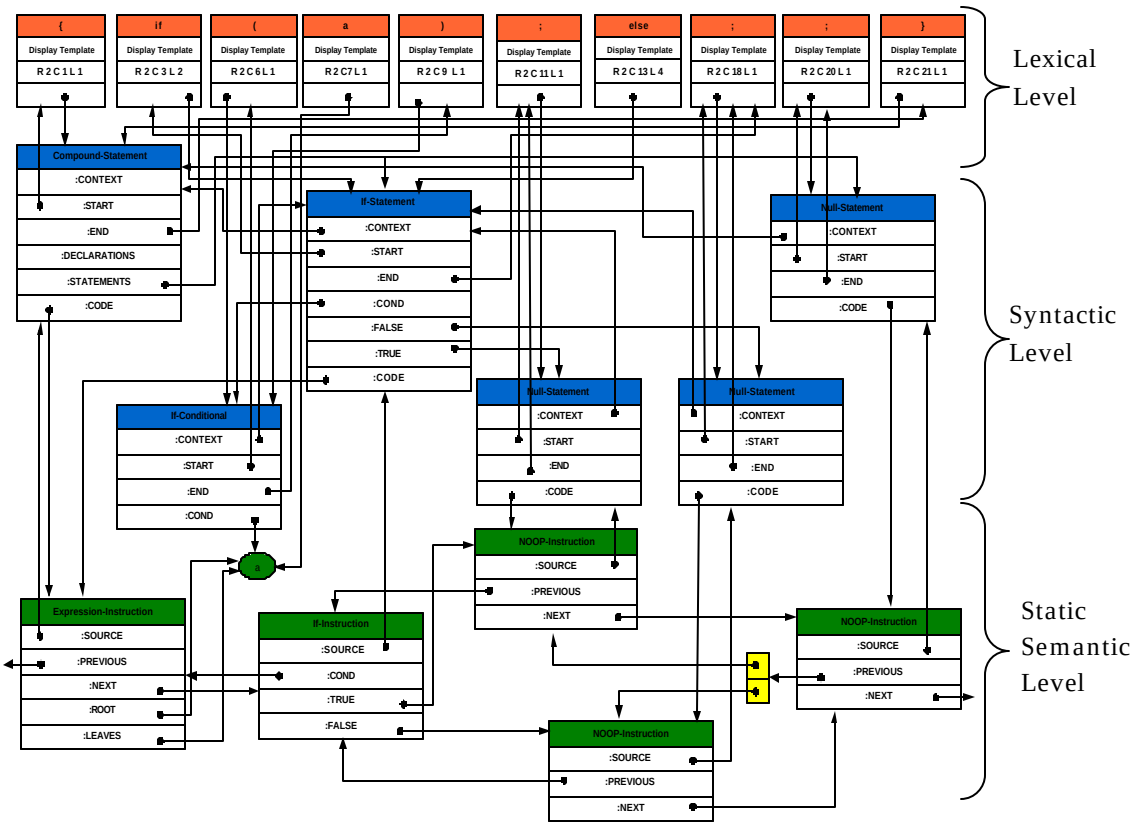

Figure 2: Representation of fixed data structure of { if (a) ; else ; ; }

4.1.1 Character and Lexical Level: Linked List

The first level is simply a doubly linked list representing the sequence of language tokens comprising the program. This sequence of tokens is a complete lexical representation of the program. It includes tokens for blank lines, line continuation characters, comments, pre-processing directives, and all the tokens comprising the program code. Each lexical token not only contains the sequence of characters determining how the token will display on the screen, but inherits

routines from its class describing how to display the token, what colors to use, etc. In addition formatting information is stored in this level, that is, each tokens source line and column information. This level is the entry point for the user interface.

4.1.2 Syntactic Level: Tree

The second level is the source code syntax tree. Each node in the tree is typed and bears an inheritance relationship. The tree is also doubly linked. Each node has a link to its parent as well as all its child nodes. This way we can freely move around the tree no matter where in the tree we may start. The user interface can directly access any node. Each node in the lexical chain has an up-link to the lowest level syntactic structure in which the token participates. Because each lexical token also has screen-positioning information the code looks like it has a text based editor user interface. One hop up and the syntactic structure is available.

4.1.3 Semantic Level

4.1.3.1 Static Semantic Level: Graph

The third level is an executable control flow graph. Nodes in the AST are of two types. The first comprise control flow structures. The second comprise executable expressions. Each control flow structure will be traversed by the tree parser to construct interconnecting links between the executable expressions. Conditional expression processing nodes will be added to the execution graph as determined by the control flow structures. This will form an executable representation of the program. Each node in the graph will be linked both in the forward and backward direction. In addition each node in the AST will have a bi-directional link with the corresponding node in the CFG. This will allow complete movement starting from any point in the program to any other point via lexical, syntactic, and semantic orderings. This level can be traversed as a static structure. Nodes can be visited forward or backward along *static* execution paths. Here no accessibility of program values is available. This represents the *static* execution connectivity. Not all these paths may be feasible under actual execution. The feasibility of execution paths is determined by the solvability of the accumulated constraints along the conditionals on the path.

4.1.3.2 Dynamic Semantic Level: Symbolic Execution

This level represents all the information that only comes into expression during program execution. This part of the third level is accessed via an *open symbolic interpreter* (described in Section 4.4). The interpretation is performed symbolically so the entire execution space (within memory limits) is represented instead of a particular or small set of individual execution states. This allows more comprehensive questions to be asked. The level is reified by traversing the CFG with a symbolic interpreter that can process navigation whether the conditionals are resolved or not. This is one of the bridges to the abstract level.

4.1.4 Abstract Level: Virtual Graphs and Structuring Memoization

The fourth level is virtual. The three levels above represent the entire fixed information content of the program. However, because of the need for efficiency and abstraction this is not sufficient. This fourth level allows data structures to be built virtually and possibly reified via memoization. It is reified by Prolog-like rules that specify how to move through the AST or the CFG. In the case of virtual data, the data is never actually extant, but exist as parts gathered logically through deduction or through interpretation. This abstract gathering can occur over the AST, the CFG, elements exposed through the interpretation of the

program or some combination thereof. This is done by rules controlling traversal of the data structure. The particular nodes of the AST or CFG inherit, through their class membership, sets of rules describing how to walk the node or how to execute the node. This way the nodes or results of executing the nodes can be visited by the search routine binding to the pieces it is looking to find or relate. In the case of structuring memoization, these bits of data will be captured in a data structure. Depending on the usage, this data structure can be a set, stack, queue, tree, or graph. It can be persistent or temporary. This structure can also be passive or active. Active data structures are used to directly guide search paths whereas passive data structures are just searched. An example of an active data structure would be to stack nodes discovered during search then in a later phase popping the nodes off and using them to direct the course of search. In Section 4.3.1 below an example of rule use also illustrates an example of memorizing persistent, passive graph. This is the case of creating a control flow graph from querying the AST. Another instance would be creating a static call graph. In this case the graph is kept around and used as for memorized lookup. The rules can also express execution at different levels. The simplest is concrete execution. This mode acts as an interpreter, which directs execution of the code as the compiled version would. The rules also will contain information on the computation of

values. This is implemented as a set of generic functions to produce and compute on values that the reduction of rules will cause to execute. This set of functions can easily be altered to work with not only concrete values, but also symbolic values [38, 92], lattices and domains [41]. This way the interpreter can be scaled up to perform symbolic execution or abstract interpretation.

The rules can describe hopping from one syntactic element to another with functions to carry out the passing of values. Any point in execution is immediately related to a position in source code along with values and program conditions. This will allow for a natural user interface as the elements are in terms of entities that the programmer directly relates to.

This entire three level structure can be “unzipped” at some lexical point and new text entered. The text will be automatically tokenized and the parser will parse the new tokens integrating the new sub-parse tree into the structured program and “zip” it back up. This allows all the search machinery to be used interactively during development, and invariants, style, and correctness can be enforced at composition time.

4.2 Reparsing

A standard LALR(1) parser generator has been enhanced to accept a grammar where the grammar rules specify a class based constructor to use to create the AST node when the production reduces. The parser generator uses this information to augment the exit arcs for the pushdown automata such that in addition to the parser accepting the normal sequence of tokens to trigger a reduction to create a node in the AST, AST nodes can be interspaced within the token stream to also be matched. This way if a line of source code is modified, only that line of text needs to be tokenized; the AST is spliced, creating a small list of AST node objects that surround the tokenized text. This much smaller sequence of objects (representing large parts of already parsed source code) and tokens can then be parsed to quickly recreate the modified parse tree.

4.3 Epitaxic Deductive Retrieval

Epitaxic deductive retrieval is based on the concept of using a single integrated data structure for all levels of source code representation, building searchable data structures on the fly through deduction, a single language for specifying program queries, and a single processing engine for all source code queries. The data structure is the virtual abstract semantic graph described in the

Section 4.1. The language is *Epitaxis* using Prolog-like predicates. This processing engine is a unification-based search routine in which queries are specified as predicates.

The search engine operates by traversing the virtual ASG trying to unify with nodes in that structure. The order and path of search, and the exact pattern to be unified with are all under programmatic control. These can be pre-determined or modified by what has already been found during an ongoing search. The search successfully terminates when the search structure is fully resolved, at which point the search goal is returned. If no such final resolution is possible, the search fails and a NIL value is returned. Using the search patterns and the unification variables the search can find parts by looking out in all directions along all four levels relating the pieces in arbitrary ways. Some of the pieces to be search upon or searched for may be constructed by the search process itself. *Epitaxis* can query a program's AST and retrieve its CFG.

4.3.1 Search Specifications

For a search to take place two things are needed, a structure to search and a search specification, which can be a search template, a search predicate, a search function, or a search rule. Search predicates and search functions are used

to compose search templates enforcing a control structure on when, where and how the search patterns are sought. All searches are assumed to have a starting position in the AST or CFG. This concept is similar to the *this* parameter in object-oriented programming. Successful searches update this position. Table 1 shows examples of elements of the search language and these are explained below.

Table 1: Elements of the Search Language

1) Unification Variable	`?label`
2) Unification By Indirection Variable	`?&variable`
3) Compound Unification Variable	`?(identifier function-name :NAME ?"API_.*")`
4) Search Template	`$(:UP ?(iter iteration-conditional) (:SLOT :COND))`

5) Search Predicates and Function	`(DEFINE-SEARCH find-undeclared-vars (?name)`
	`  (PROG1 $((:IN $(function-declarator FAIL))`
	`  ?(var variable :NAME ?name))`
	`  (NOT (var-declaration ? ?var ?name))))`

6) Rule Based Search	`(--->(COMPILE-STATEMENT`
	`  ?(if-stat C:if-statement`
	`  :COND ?(if-c C:if-conditional :COND ?cond)`
	`  :TRUE ?(true C:statement :CODE ?&t-first)`
	`  :FALSE ?(false C:statement :CODE ?&f-first)`
	`  :CODE ?&c-first)`
	`  ?prev`
	`  #H(?t-last ?f-last)`
	`  ?next)`
	`  (AND (COMPILE-EXPRESSION`
	`  ?cond ?prev ?c-first ?c-last ?if-instruction)`
	`  (COMPILE-STATEMENT ?true ?if-instruction ?t-last ?next)`
	`  (COMPILE-STATEMENT ?false ?if-instruction ?f-last ?next)`
	`  (IS ?(if-instruction C:if-instruction`
	`  :COND ?&c-last`
	`  :TRUE ?&t-first`
	`  :FALSE ?&f-first)`
	`  (MAKE-INSTANCE C:IF-INSTRUCTION :SOURCE ?is)) ))`

7) Execution Based Search	`(===> (EXECUTE-EXPRESSION #I(CLASSES-C::conditional-expression`
	`  :COND ?cond`
	`  :TRUE ?true`
	`  :FALSE ?false)`
	`  ?value)`
	`  (IF (EXECUTE-EXPRESSION ?cond ?c-value)`
	`  (EXECUTE-EXPRESSION ?true ?t-value)`
	`  (EXECUTE-EXPRESSION ?false ?f-value))`
	`  :VALUE (EXECUTE::QUESTION-MARK ?c-value ?t-value ?f-value))`

A basic building block of searches is unification variables, which are not searches by themselves, but are used within search patterns. A simple unification

variable is the `?` symbol immediately followed by the variables' name, as shown in the first example. Compound unification variables allow for restriction on the class of the object that can unify with it. It is also possible to specify constraints on the slots within the object. The third example above is a complex unification variable to find all function names matching the regular expression, "API_*". The unification variable is named `identifier`. It can only unify with an object of class `FUNCTION-NAME` and the `:NAME` field must match the regular expression "API_*". A search template has three parts: a direction(s) of search; a target to match; and an action to perform if the search is successful. The first specifies which branches of the tree to look along given a starting point, for example, up the tree to the parent node or down a sub-tree, or nodes of a specific type, etc. The second part specifies what to look for. It can be some specific object, an object that is an instance of some class, or an object that has certain specific properties amongst its slots. The last part specifies something to do if the search pattern is successful in finding an instance of the pattern. This usually specifies movement in the search tree relative to the matched node, such as down one of its slots. The fourth example in Table 1 shows a search template. Given any point within the body of any type of `ITERATION-CONDITIONAL`, this search template will find the corresponding conditional clause. Starting from some position in the

AST, it will look `:UP` the parent links trying to unify with a compound unification variable to find an object of type `ITERATION-CONDITIONAL`. (Note that via inheritance this will match a `FOR-CONDITIONAL`, a `DO-CONDITIONAL` or a `WHILE-CONDITIONAL`, but not an `IF` or `SWITCH-CONDITIONAL` which inherit from the class `CONDITIONAL` but not `ITERATION-CONDITIONAL`). If one is found, the current context pointer moves to the conditional field of the matched object.

Search predicates are the built-in primitive operations of the search language, and are based on the built-in predicates available in Prolog. They allow the search patterns to be composed, and include simple directives such as simple sequencing of search patterns, a wide range of controlling search based on whether other searches are successful or not, selection of search alternatives based on the returned type of a previous search. The complexity of the search predicates and consequently power of the searches go beyond Prolog because the search is performed on a syntax tree, and not just a list of facts. Furthermore, the search can use information about a node's behavior and semantics, not just its identity. Finally, search functions allow user-definable search predicates. A sequence of search patterns, search predicates, and search functions can be

arbitrarily composed to create sophisticated searching control structures. Search functions can call other search functions, so recursion is also possible.

The fifth example presents a search function that searches for variables with no declaration. The built-in search predicate, `PROG1`, indicates that sub-searches should be sequential, meaning it starts the next search where the previous one completes, but ultimately returns the value of the first sub-search (hence the `1` of `PROG1`). The first sub-search looks inside current context for a variable, but will fail if it tries to look inside a `declarator`. Once this variable is found, the second sub-search is initiated by calling a search function to find the variable's declaration. But since that search is embedded within a `NOT` predicate, the search declaration has to fail for the `NOT` predicate to succeed. This will cause the `PROG1` predicate to succeed returning what was found by the first sub-clause of the `PROG1`, i.e. the variable that has no declaration. The function is defined with a parameter. If this parameter is bound at call time then the search will only look for undefined variables with the given name. If this parameter is unbound at call time then the parameter will become bound to the name of the variable that was found. The search saves its context, so it can be called repeatedly to find all the undefined variables.

The sixth example shows a rule used in rule based search. It also uses *unification by reference*. This rule builds the CFG of an `if` statement from the C language. By convention, the first argument supplied to the predicate is the current node in the structure being searched, in this case the AST. This argument and the other three are bound by the rule that triggered this one. Typically `?prev` will be bound by the "compilation" of the previous statement and `?next` will be unbound but will be used as a holder for the last instruction created in this rule "promising" to tell where to link to for the next instruction. The `#H(?t-last ?f-last)` represents a hyper-edge and will eventually be bound to the next instructions previous field. It is a hyper-edge because the previous instruction to the next instruction could have come from either the last instruction in the true clause or the last instruction of the false clause. If either of these contains a branching instruction then the corresponding `?t-last` or `?f-last` will be unbound and the hyper-edge machinery will remove it. The body of the rule is responsible for building the instructions composing the `if` statement. There are four predicates within an `AND` clause. These all have to resolve or all the bindings made so far will be undone and the system will try another rule. This might also happen if the head fails to unify. It will happen if the particular `if` statement does not have a false statement. The rule will fail

because the compound variable `?(false C:statement :CODE ?&f-first)` will not unify with NIL. Assuming the head (LHS) does unify with the AST node, the retriever will try to resolve the four sub-assertions. The first one will process the conditional expression; the second one will process the true clause statement; and the third one will process the false clause statement. The last is the built-in `IS` predicate. As in Prolog, this predicate evaluates its second argument and tries to unify it with its first argument. In this case an `if-instruction` node is created, binding the :SOURCE slot with the value of the `?if-stat` variable which was obtained when the head of the rule was unified. The `IS` predicate mandated unification will cause bindings between the variables and the slot locations within the `if` instruction node. When these variables bind to values these slot locations will also take on the value because of *unification by reference*. Note also that when the head of the rule is matched, it does destructuring, binding the variables to the corresponding AST node structures within the `if` statement. These variables are passed to the body (RHS) predicates used to "compile" these corresponding parts of the `if` statement. The `?&t-first` and `?&f-first` variables create bindings to the slots within the :CODE fields of the statements within the if statement. This way when the COMPILE-STATEMENT predicates for the statements eventually build instructions for

these statements, the first one will get stored in the :CODE slot for the source AST node. In a similar manner, the retriever will call rules walking the entire AST and in the process of this "search" instruction nodes will be created and the variable bindings will insure that all the nodes links are properly linked up creating the CFG of the AST. This CFG can later be queried/executed to find additional facts about the AST that spawned it. This is described next.

Another kind of search specification indicated in Table 1 is execution-based search. This type of search allows the search engine to visit nodes in the order in which they will be executed. Here the search engine walks the CFG instead of the AST. This search is aided by an *open* interpreter (described in Section 4.4). This type of search is necessary to perform dynamic analysis and symbolic execution. This search specification can express execution at different levels of abstraction. The simplest is concrete execution. This mode acts as an interpreter, which directs executing the code as the compiled version would execute. The interpreter, in addition to following the CFG, is comprised of methods, which implement the runtime system. These methods are called to implement data creation and operator evaluation for the language being interpreted. This set of methods already works with concrete values and

symbolic values and can easily be overridden to work with abstract values such as lattices and domains [41]. This way the interpreter can be scaled up to perform abstract interpretation. Because of the inter-level linking, any point in execution is immediately related to a position in source code along with values and program conditions. This will allow for a natural user interface as the elements are in terms of entities that the programmer directly relates to.

4.3.2 Search Engine

In *epitaxic deductive retrieval*, the search engine uses the same concepts and principles as Prolog, but applied to a tree or graph data structure rather than a list of facts. Matching occurs through unification of variables to nodes in the graph. Data values are bound to unification variables or AST node slots for output. When sub-parts of the search fail, variables become unbound, so that other matches can be found. Backtracking is used to unbind variables when sub-searches fail or the tree needs to be searched exhaustively. These concepts as they apply to *epitaxic deductive retrieval* will be described in more detail below.

The search specifications and the virtual ASG become linked via unification variables. Unification variables are allowed within the elements of the search specifications. An unbound unification variable will become bound to a

part of the tree that matches the search target. A bound unification variable must be equal (EQL in LISP parlance) to a corresponding part of the tree structure or else the search fails. When branches of the search fail, all unification variables that were bound during that search are automatically unbound. Then as the search tries other paths through the tree, new bindings are made. These unification bindings can be passed as parameters to search predicates and search functions allowing what is found during one sub-search to be consulted during another sub-search. These, too, are unbound automatically if sub-searches happen to fail. Ultimately, if a search succeeds the final node of the AST that is matched is returned from the search. In addition, any unbound unification variables passed as parameters to the initial search may now have bound values corresponding to parts of the search tree located along the way to the final result. As in Prolog, passed unification variable parameters are bi-directional. If they are bound when calling the search function, they hold values input. If they are unbound when calling the search function, they may become bound and hold values for output. The bindings may be indirect, that is, a unification variable may become bound to several unbound unification variables until one of them finally becomes bound to some part of the search tree. Any restrictions on each of those unification variables are all enforced.

Each pattern within a search form specifies its search field and any restrictions within that search field. The search field defines what part of the tree or graph will be searched. Each search field type also has an implicit search order within that field. Restrictions can limit which class of nodes will be searched from a given point in the search field. Also, for a given class, which slot or slots should be searched next can also be specified. It is also possible to indicate a search failure if a particular class is encountered. These restrictions can be specified to occur at arbitrary depth of link traversal.

Search functions define relations between their parameters. If the search is successful, then the relation specified by the function exists, otherwise it does not. Because some or all of the parameter can be left unbound when the function is invoked, the search function effectively goes looking for objects that will satisfy the relation specified by the function. Run singly the search will find the first instance that satisfies the function. But, since the search state is saved, and the algorithm admits backtracking, the same search can be run until the entire tree is exhausted to find all instances that satisfy the function. Using the search engine, the search tree (e.g. the AST) becomes a database upon which arbitrary logical relationships can be also found, verified, and/or extracted.

There are six possible levels of search. These correspond to the levels within the data structure representing the program. They are 1) along the token linked list, 2) along the AST, 3) along the CFG, 4) “logically” by following Prolog deductive retrieval like rules, 5) dynamically by following the interpreter, and 6) abstractly by creating structures via memoization and following pathways in there. Since all the data structure representations of the program are interconnected, the search engine can switch levels of search as needed. Brief descriptions of these six levels of search are given below.

4.3.2.1 Lexical Search

In lexical search, the search engine simply walks the liked list of tokens, either forward or backward, looking to match either on the class of the token, and/or the name of the token using regular expressions.

4.3.2.2 Syntactic Search

In syntactic search, the search engine walks the AST. Here, nodes are matched by class membership and structural relationships. This form of search is complicated by the need to express which direction(s) through the tree to search.

4.3.2.3 Control Flow Graph Search

In static semantic search, the search engine walks the control flow graph. The search can be configured to look down only true branches, only false branches, all branches, or randomly chosen branches.

4.3.2.4 Logical Search

In logical search, the search engine "deduces" the search order via predicate calculus like rules. Given some point in the program, using the nodes' class, the search engine looks for a rule with a matching left hand side (LHS). If one is found, the right hand side (RHS) contains specifications for the continuation. This process continues as the RHS typically contains further rules. Logical search is the foundation of the search methods listed below.

4.3.2.5 Symbolic Execution Search

In execution search, the search engine consults the symbolic interpreter as a co-routine to determine where to go next. As the interpreter can process symbolic values and consequently conditionals may not be resolvable, the search path is a multi-pathed execution tree moving breath first. This is described in more detail below in section 4.4.

4.3.2.6 Abstract Search

In abstract search, the search engine will search along a data structure build by another search via memoization.

4.4 Symbolic Interpreter

The search engine makes use of an *open* symbolic interpreter. Sometimes in the implementation of a compiler, the parser and the lexer are organized as co-routines. Each run with their own internal state. Every time the parser needs another lexeme it suspends itself and calls the lexer, who then resumes, finds another lexeme, suspends itself, and gives the lexeme to the parser, which then continues with the new lexeme. In a similar manner the interpreter and the search engine are organized as co-routines. Every time the search engine needs another node to search it suspends itself and calls the interpreter. The interpreter then resumes its state, advances the state of execution, suspends itself, and returns control back to the search engine. The search engine now has access to the next state in the running program's execution, including the current position in the control flown graph, the corresponding syntactic structure that produced the node in the CFG, and any value(s) that were input and are output by the executing node in the CFG. All of these are available as targets of the search.

The interpreter operates on a control flow graph. In other words, a CFG is an input (a program) to the interpreter which executes it. The symbolic interpreter is comprised of six components. These components are 1) a set of rules that walk the CFG and trigger the methods, 2) a collection of methods, 3) a set of typed values, 4) a symbolic memory management system, 5) a propagation of constraint system, and 6) a symbolic execution tree and the control system which grows the tree.

4.4.1 Set of Rules

For each language that the system can work on a set of rules needs to be written which describe how to walk the CFG and what to do as the nodes are traversed. These rules are interpreted by the search engine. Instead of doing *logical* deduction with the rules the engine does *structural* deduction. The engine tries to match a rule head to a node in the control flow graph. If the rule head and the CFG node unify, then the body of the rule is executed. This typically causes an advancement through the CFG. Rules matching expression nodes have clauses causing methods to be invoked on the return values of the rules which traversed the expressions arguments, thereby executing the expression. The rules for C are detailed in Appendix A.

4.4.2 Collection of Methods

The body of the interpreter is a collection of generic functions. For the initial prototype they comprise all the operators of the C programming language and a set of system functions such as `malloc`, `free`, `setjmp`, `longjmp`, etc. The generic functions have predefined methods for standard data types of C such as numbers, characters, and addresses. In addition there are predefined methods for two forms of symbolic values (proxy values and interval sets) and error values. It is also possible to define additional classes of values and to define methods that operate on instances of those classes. The generic functions select on all arguments so you can define different methods for the `+` operator which takes an integer and a float versus an integer and an address versus two integers. You can also define abstract classes such as `PARITY` with instances `EVEN` and `ODD` and define methods for `+` which operate on type `PARITY`. The constant load operator can be overridden to convert integers to their proper `PARITY` value.

4.4.3 Set of Values

There are several different types of values that the methods described above and hence the interpreter can process. These are expandable using the object system. The staple of the system are concrete values which are the

standard data types of C. These include different size and types of numbers, characters, and addresses. In addition, the system recognizes symbols which represent special values such as "`UNINITIALIZED-VALUE`", "`UNSPECIFIED-VALUE`", "`ILLEGAL-VALUE`", "`NULL-DEREFERENCED-VALUE`", etc. These are used in part to make the operators closed. That is all operations will return a processable value no matter what input. Although, beyond indicating an error these values are not very useful. Passing "`ILLEGAL-VALUE`" as an argument to any operator will result in "`ILLEGAL-VALUE`" being produced. The one big exception to this is "`UNSPECIFIED-VALUE`". This triggers symbolic values. The system handles three types of symbolic values. The simplest are interval sets. These represent arbitrary collections of intervals of numbers. Arithmetic can be performed on them returning modified collections of intervals. There are also proxy values and proxy addresses. Proxy values represent unknown values which can later become concrete or an interval set. Proxy addresses represent *logical* segments of memory. The content may or may not be known; what they really represent is that the location of the segment of memory is unknown and its size may also be unknown. A proxy value is created whenever a variable is read whose value is not known. This happens whenever execution starts after the point in which the variable was given a value such as executing a function

without a calling context. The function arguments will be unknown. Proxy values are created as stand-ins for accessed but unknown concrete scalar values. Proxy addresses are created for accessed but unknown pointer or compound values. If a proxy address is dereferenced, proxy memory is created for it to point to. Proxy values or addresses may become concrete. This can happen when conditionals are assumed as described below in Section 4.4.5.

4.4.4 Symbolic Memory Management System

The memory management system is responsible for accessing and storing values for all variables, heap memory, and proxy memory. In addition, it must keep track of the different values they may have under different program paths. Values which are set in ancestor paths are shared by all sibling paths unless one overwrites it. That path and all its siblings will share the new value, while the other paths still share the old value.

The system is implemented using a compound interval tree. The outer tree is keyed by the address segment. The inner tree is keyed by the path the value is current in. Paths are represented by intervals. The root path is the largest interval, sibling paths by sub-intervals of their ancestors.

4.4.5 Propagation of Constraint System

The propagation of constraint (POC) system is used when determining the consequences of assuming conditionals to be true or false or assuming switch case selection. This system is modeled after that described in [1]. Whenever a conditional is executed and it is not possible to determine its value because it contains symbolic value(s) a POC system is built for the conditional and it is processed for both possible truth values or all possible case values. The addition of the assumed conditional value can be enough to determine the value of some or all of the symbolic values. For each equivalence class of values which determine a possible conditional value another program path is spawned. As an example, assume that `x` and `y` are variables with symbolic value in the conditional `if (x == 0 && y != 1)...` Since the values of `x` and `y` are unknown the condition cannot be resolved so the POC system will assume `(x == 0 && y != 1)` to be both true and false. Under the assumption that it is true `x` will have the value `0` and `y` will have an interval set value of {(-inf, 1), (1, inf)}. These values will be set in memory for the true program path. Under the assumption it is false two different possibilities exist: 1) `x` = {-inf, 0), (0, inf)} and `y` is still an unknown symbolic value 2) `x` is still an unknown symbolic value and `y` = 1. In this case two different false paths will be spawned with their memory

set accordingly. This `if-conditional` will spawn three paths total with the corresponding modifications to memory. If the POC system can prove that no solution exists for an assumed conditional value, the path is *infeasible* and is terminated.

The system also remembers which variables have had a proxy-value assigned to it so if the POC system assigns a value to it, all the other variables that were assigned the proxy-value will also get the assumed value. As an example assume the following code exists with `y` a symbolic value:

```
x = y;
if (y == 0) ...
...code involving x...
```

The `if` statement will spawn two paths. `y` will be 0 on one and not 0 on the other. However, the system will also know that `x` will be 0 on the path where `y` is zero and vice versa.

4.4.6 Symbolic Execution Tree

The interpreter can execute multiple paths of an executing program. Because the interpreter can operate on symbolic values, conditional statements

may not be resolvable. In this case the interpreter forks another path and continues with both executions, one assuming the condition is true and one assuming the condition is false. This creates an execution tree. The leaves are all the currently active program paths and the internal nodes are all the decision points. The tree is not necessarily binary. Switch statements generate a branch for every possible (feasible) case including default. In addition if-statements can also generate more than two possible branches. If the condition includes an AND operator with both symbolic arguments then in addition to both arguments being non-zero to generate the true branch, there are two possible false branches, one for each argument independently being false. For the OR operator there is one possible false branch, all arguments being false and two possible true branches, one for each argument independently being true. If ORs and ANDs are combined then it is possible to have multiple true and multiple false branches for one if-conditional.

The nodes of the execution tree are used as an indexing scheme to remember what CFG nodes have been executed under what path. This allows tracking of how many times a particular path has stepped on (executed) a particular node in the CFG. This can be used to determine how many times

statements within a loop get executed and stop paths at a predetermined number.

4.4.7 Collecting Semantic Information

Using menus attached to items in the AST, the user can select *collect points* where dynamically produced information will be collected from. The interpreter currently supports collecting information from three types of locations. These are: 1) collecting information about values written into `struct` and `union` members, 2) collecting information about function return values, and 3) collecting information about function parameter values. In addition to collecting information at these points, the user can attach assertions to be executed at these points to verify values. Finally, function return values and function formal parameter values can have *tracers* attached to them. When a function returns a value, or an actual gets bound to a formal parameter, these values are marked by the function name or the formal name respectively by the *tracer*. This enables *collect points* to know where the value reaching it came from. This can be useful to determine if a value being stored into a `struct` member has been processed by some particular function.

The collectors can be set to collect information derived from the value as it passes through the *collect points*. The value itself can be collected (useful for members that get assigned `enum`s), assuming a `struct` value, a value within a member of the value can be collected, the type of the value can be collected, a label attached to the value by a *tracer* can be collected, or a value produced by an arbitrary filter function on the value can be collected.

The information is collected as a structured map of objects to attributes. The actual `struct` data constitute the objects and the set of members and data values written constitute the attributes. These maps are processed using *formal concept analysis* [67] to create a partial order on the equivalence classes of information collected. The resulting concept lattice can be very revealing about how the program uses data. Examples of this are how `union` member access is related to a type field within the `struct` or the structure of function call signatures. This type of information can be used to help understand how to restructure a program to be more object oriented.

4.5 Prototype

A prototype of *Epitaxis* has been implemented to conduct this research. *Epitaxis* is embedded within an augmented LISP [127] environment. It has a large

complement of standard LISP features such as a subset of the Common Lisp Object System (CLOS) [118], a package system, hash tables, generators, gatherers, etc. In addition there are many non-standard LISP features such as support for advanced data structures such as b-trees, interval trees, union-find-deunion, tries, and suffix trees. There is an embedded Prolog system, an L system, sequence and tree alignment algorithms, various machine learning algorithms, a lexical analyzer generator and an LALR(1) parser generator. These have been combined to form a platform upon which this research was conducted.

LISP was chosen as a foundation for this research for the following reasons. First, the LISP environment is very robust. It is the second oldest programming language and has been under evolution for over fifty years by very bright people evolving it to solve very difficult programming problems. The environment is interactive and reflexive, designed for exploratory evolution of complex systems. LISP also has a very robust object-oriented system called CLOS. CLOS has a *meta-object protocol* [87], which makes it reflexive, flexible and very powerful. There is a synergy of parts based on its hybrid structure. The system is implemented in C, which makes it efficient. All the heavy pieces of machinery are directly implemented in C and given an outer wrapper to be

callable from LISP. The C code can directly call modules within the interpreter in C for efficiency, but all the pieces have LISP interfaces so they can be called interactively. In essence, LISP supplies the structure of the environment, a large library of subsystems, and a very sophisticated scripting language, which can be used for interactive unit testing.

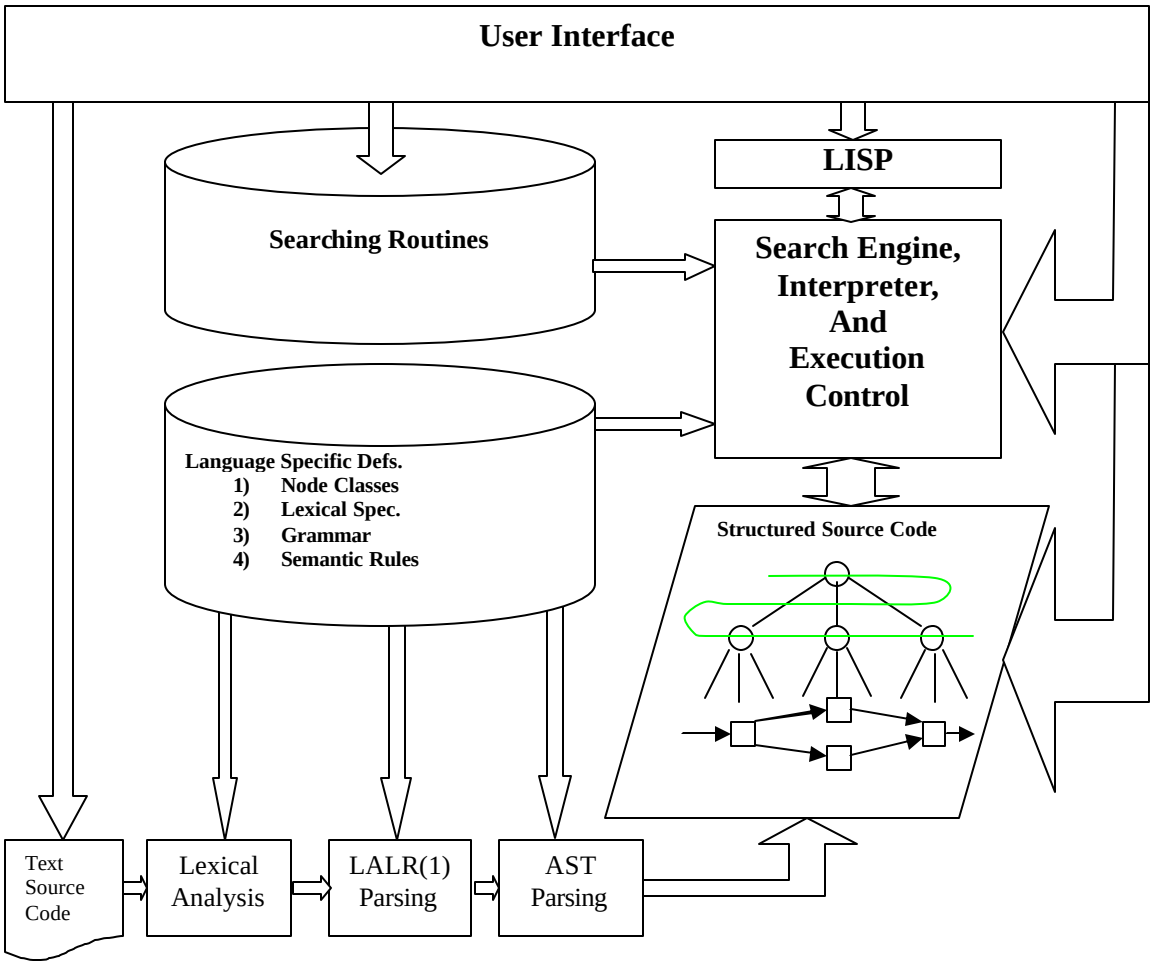

Figure 3: Architecture and Processing Pipeline for the Epitaxis Framework

Figure 3 shows the basic architecture and processing pipeline of the system. The user interface allows the loading of an ASCII source file. If this language has not been previously seen a lexical analyzer and parser are built using its database of language specifications. The ASCII source code is then converted into its AST and ASG. The user interface now has access to the semantics of the program through menus associated with the lexical and syntactic items in the AST. These can trigger the predefined searches associated with the objects in the AST. It is also possible to build additional search routines through the user interface and trigger those queries as well. The results of searches are expressed by highlighting parts of the AST or by building other views, which are "hot" clickable allowing navigation to the corresponding part of the AST. The user interface also allows direct typing of characters to alter the source code, which reparses the appropriate parts of the AST once the current line is exited. In addition structure based alterations of the AST are possible by dragging and dropping syntactic items. Having an editor type interface to the query system and interpreter allow a high bandwidth interaction with the program.

"In theory there is no difference between theory and practice, in practice there is."
--Yogi Berra

Chapter 5

5. Findings

Epitaxis gives the programmer a language in which to ask questions of software and supplies answers. In this chapter we will look at some of the kinds of questions *Epitaxis* can answer and how well it can answer them. The first issue is how expressive and relevant are the questions that *Epitaxis* can answer. Some of the kinds of questions that *Epitaxis* is designed to answer are questions that came up while implementing *Epitaxis.* There is also the issue of efficiency, that is, was the answer returned in a timely manner. Lastly there is the issue of query range. Some of the questions answered by *Epitaxis* can be answered by existing

query systems and some of them cannot. No system that we know of can answer as wide a range of questions.

5.1 Representation

Before *Epitaxis* can answer any question about software it must first build a transformation of the program text. This is the first factor affecting scalability. How much memory does it take to represent a realistic size program and how long does it take to build it. We let *Epitaxis* compile (build the AST and the CFG) its entire code base to see how much memory it consumes and how long it takes to load. The LISP system in which *Epitaxis* is embedded is 392,690 lines of C code representing 12,180 functions in 205 .c files and 120 .h files. All of this needs to be in memory for *Epitaxis* to perform syntactic search on the entire code base. However, semantic search (or syntactic search for that matter) really only needs the files being searched in memory, not the entire code base. *Epitaxis* took 88 seconds to scan and parse the system and consumed 1.3Gb bytes of memory. It took an additional 86 seconds to produce the CFG (only needed for semantic search) consuming an additional 97 Mb of memory. The memory figure includes both the memory needed to hold the representation and the memory consumed in the process (which would be recycled if the garbage collector was turned on).

Visual Studio 2005 took about 60 seconds to compile the same code. Averaging out these numbers on a per file basis gives 7.1 Mb of memory to hold a 1915 lines of C code file, compiling in 0.85 seconds.

5.2 Syntactic Search

Syntactic questions involve the relationships between parts of the syntax tree. These types of questions pertain to navigating code, that is, finding some static location in the code base: some point in the AST. They also pertain to finding structural abnormalities in the AST such as finding all variables that do not have a declaration or finding all loop structures that do not modify at least one of their loop conditional variables: something that is not in the AST but should be there. The syntactic search language of *Epitaxis* is expressive enough to answer any syntactic question. Although it is possible to give precise answers to syntactic questions, in practice there are two difficulties with this. The first is that it may be difficult to express the exact syntactic relationship that will answer your question. You will get what you asked for, but it may not be what you wanted. The second is that the question is really a semantic question and the syntactic version of it is only an approximation of what the programmer really wants to ask. You will get what you want but not all of it. Assuming no bugs and

that you are asking the question you really want, *Epitaxis* has perfect precision and recall on syntactic questions.

Epitaxis advances syntactic search in two ways beyond what currently exists. The first is that the syntactic search engine is not just a finite state machine but is a pushdown automata. This increases its expressive power (at the cost of a more complex query language). An example of a search involving needing a pushdown automata and not just a finite state machine is finding all places where a `struct` member is referenced. It is not sufficient to find all the member access expressions with the same name because they can belong to different `struct`s. You have to know the type of expression that the member applies to. The difficulty is that this can be nested such as `x->a->b->c.` To know what `struct` member `c` applies to you first have to climb down the expression tree to `x`, stacking the member names as you go since these will be needed later. Once at `x` you find the declaration of `x`, then find the actual `struct` definition (which may involve looking through several `typdef`s). Now search the `struct` definition for the declaration of `a` (we popped `a` off the stack of the member names we collected as we walked down the expression tree). Now lookup for the `struct` definition of the type that `a` points to, search its declarations for one

with member `b` (again popped off the stack). This continues until the stack is exhausted and we found the declaration of member `c`. All other member references to `c` have to be checked this way to find the actual declaration within the `struct` definition to compare if it matches the original. So the search is not just walking the syntax tree as a finite state machine can but requires stacking states to refer to later to guide the search. This type of search is very common. If a programmer is trying to understand code to enable him to modify a `struct` definition he often needs to know all places that some members of that `struct` are referenced. It is helpful to not have to weed through all the false positives a simple match on the member name will make, especially for a very large software system with many `struct` definitions.

The second advance that *Epitaxis* makes over other syntactic software query systems is that it can not only find a single point in the syntax tree related to another, or tell you that a point within the tree does not exist, it can extract an entire structure related to the syntax tree (or some part of it). An example of this is querying the syntax tree and retrieving the control flow graph. Not only is the graph retrieved, but it can also be bidirectionally linked to the nodes in the syntax tree related to the nodes in the control flow graph. This area of search has

not been further pursued, however it may be possible to use *Epitaxis* to retrieve a *dominance tree* or some other structure related to the syntax tree.

Several syntactic searches were performed on the entire LISP code base. The results are summarized in Table 2. *Epitaxis* can find syntactic information on a large body of code in a an interactive time scale.

Table 2: Syntactic Search Performance on LISP System

Search	Number Found	Time (Seconds)
All Function Definitions	12,180	0.124
All Function Calls	88,581	1.235
All Variable Definitions	31,494	2.357
All Global Definitions	9150	0.2
All Unused Locals	1076	11.468
Unmodified Loop Conditionals [2]	171	0.216

5.3 Semantic Search

Semantic questions involves relationships between data values produced by executing code. *Epitaxis* can answer two different categories of questions pertaining to executing code. The simplest is about runtime singularities or bugs. These are found almost for free. As *Epitaxis* executes code it has to recognize

[2] This is an example of a syntactic search that should really be semantic. Because of function calls and pointers, loop variables can be modified without it being syntactically obvious. However, this search is useful because programmers often forget to update the loop variable.

problems such as `NULL` dereference or indexing out of bounds if for no other reason than to protect itself from crashing. These are reported to the user. There are a whole set of these kinds of problems that *Epitaxis* finds; null dereference, accessing memory out of bounds, reading uninitialized memory, etc...

The second category of questions is much more semantic in nature. Here you direct *Epitaxis* what to look for. These questions have answers that are collected over the execution space of the program. Because the system uses symbolic execution and not just dynamic analysis to search the execution space the answers have much higher recall. These are questions like: "What is the collection of values or value types that a function returns? "What is the collections of calling signatures that exist for a function?" "Did the value stored in some member location come from some particular function?" "What is the set of functions that can give a value to a member field of some struct?" "Has the value been process by function x?" "Given some struct object what is the range or structure of the relationship between values of members within the same struct?"

One of the difficulties of asking these kinds of questions is that you have to know enough about the code to ask the question, or that the questions is sensible to ask. It doesn't make sense to ask these types of questions on any piece

of code like it does to ask if a `NULL` pointer is ever dereferenced. These questions presuppose some knowledge about the code that you are asking about.

5.3.1 Basic Semantic Search - Finding Bugs

We ran *Epitaxis* in its basic bug search mode to verify that it can find bugs and to determine its scalability. Detecting bugs at their point of failure is easy if the symbolic interpreter can reach that point in the code. The important question is how efficiently can *Epitaxis* reach a high enough percent of code coverage to find bugs? Or to put it in other terms, does it scale to real world examples. With current technology, there is no way that symbolic execution can reach a significant percent of code coverage on an entire application. The execution space is simply too large. Unlike the symbolic execution systems which instrument code and have to start from the beginning (or set up drivers for particular functions), the user of *Epitaxis* can select any piece of code or function body interactively and start executing from there (missing context is simply proxied). This allows code to be searched/tested in manageable size pieces. We present results of running *Epitaxis* on three different examples: 1) a toy program, 2) a small (169 LOC) utility program, and 3) a large function (1294 LOC with 82 embedded function calls) within a large application. These examples

demonstrate the bug finding ability and the time and memory scalability of *Epitaxis*. All examples were run on a 2.62 GHz AMD Athlon™ 64 FX-60 Dual Core Processor with 2.50 GB of RAM.

We first ran *Epitaxis* on the testme() example of Figure 1. This example is reported on in [106] as taking about two minutes to find the ERROR statement using hybrid *concolic* testing with a 2 GHz Pentium M laptop with 1GB of RAM. *Epitaxis* found the error in 0.312 seconds. It explored 528 paths of length 11, consumed 19 MB of memory and executed 2,557 expressions and 9,374 conditionals. This is a toy example that is conditional heavy, designed to test symbolic string input.

A second test was run on tr.c, a GNU core utility to translate characters. This code is array pointer and array indexing intensive. The code has many loops accessing elements of strings and character arrays. The particular version used here comes from MINIX and can be found at [111]. *Epitaxis* was run on tr.c until it ran out of memory. *Epitaxis* found 7 errors within 2 seconds, 6 pertaining to accessing uninitialized memory and one index out of bounds, and ran for an additional 48 seconds until it ran out of memory. It had about 85% code coverage at that point. Six of the errors were all tracked down to the fact that *Epitaxis* doesn't understand the structure of the `argv` input parameter to `main(int`

`argc, char **argv)`. These are an array of input strings. *Epitaxis* assumes the general case where they can possibly be null strings which is not possible in `argv` pointers. Given the possibility of the strings being null, the errors are valid. The other error is genuine caused by a malformed input string. It is the same error reported in [28]. This error was found in 1.06 seconds. Code coverage grows very quickly, then tapers off growing very slowly. The problem is that once the interpreter advances far enough it is executing hundreds of thousands of paths, so the movement through code slows down to a crawl. Currently, *Epitaxis* runs breath first through the execution tree. By the time *Epitaxis* ran out of memory it was executing 336,316 concurrent execution paths. Since *Epitaxis* can keep track of how many times it has stepped on a piece of code, it should not be difficult to add a heuristic to explore paths that have not been stepped on or stepped on less times first, and use breath first when step counts are equal. This should vastly improve coverage rates and is left for future research. Graphs of code coverage and path growth are shown below.

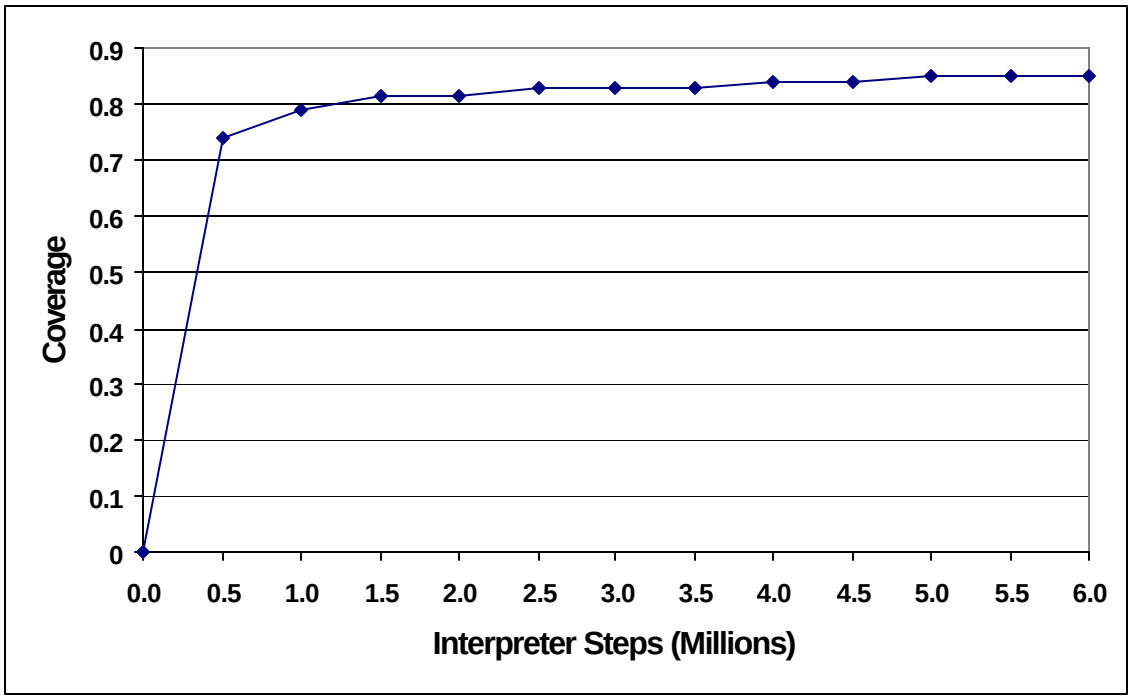

Figure 4 Code Coverage for tr.c

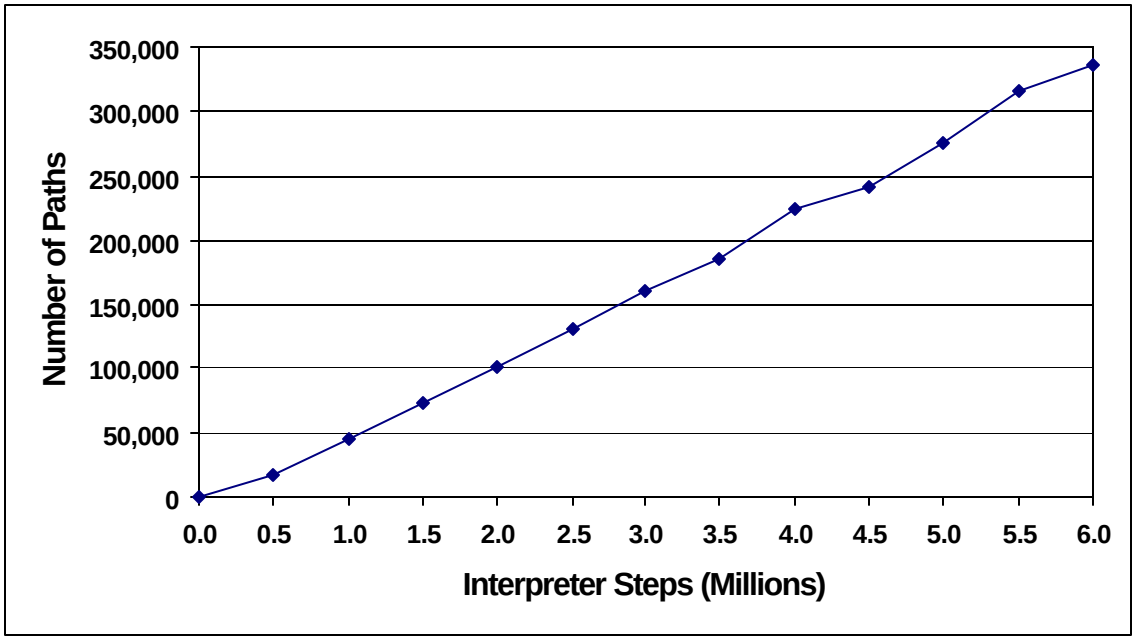

Figure 5 Path Growth for tr.c

A third test was run on the deductive retrieval function of a prolog interpreter. It is a large unwieldy function converted from assembly language to C code. The main retrieval loop function is 1294 lines of C code. There are an additional 15 small auxiliary functions included and 67 functions not included which were called by proxy (they return a proxy value of their return type). This example contains messy code, and three large switch statements which switch on the predefined prolog predicates. The code is also "pointer-to-structure" intensive.

The test was run in four stop modes: 1) stopping when a path steps on itself the first time, 2) stopping when a path steps on itself the second time, 3) stopping when a path steps on itself the third time, and 4) executing all paths without stopping. All cases were run until the system ran out of memory. The results are summarized in Table 3 and the figures below.

Table 3: Symbolic Execution Statistics executing 5 million rules on retrieve()

Mode	Running Rate	% Code Coverage	Total Paths	Active Paths	Avg. Length	Running Time	# Exprs. Exec.	Errors Found
1)One Step	8,933	53.57	138,474	80,154	31.19	561.48 s	810,245	2
2)Two Steps	9,110	91.41	183,926	129,807	25.82	551.47 s	816,081	16
3)Three Steps	5,126	88.66	187,727	123,466	25.45	975.84 s	896,333	15
4)All	94,006	68.78	265,777	258,028	31.27	54.13 s	1,093,615	10

The code contains two primary types of looping constructs. The first is looping to walk lists. This structure hurts coverage. The system goes around forever without covering new code unless the step count mode is turned on to terminate the looping. The second is the main interpreter loop. The prolog interpreter loops as it processes predicates. Some of these (such as `AND`, `OR`, `NOT`, `IF`) are predicates on predicates and create complex stack structures. Because of the predicate nesting, a large percentage of code will not be executed unless the main loop runs at least twice. This is why the code coverage gets stuck at such a low value when the paths are not allowed to step on themselves (loop bodies execute only once (mode 1). The combination of these two looping constructs make mode 2 have the best performance; it loops only enough to get coverage. Mode 4 runs much faster since it doesn't have the overhead of checking step counts.

Memory usage increases linearly per *Epitaxis* interpreter step (see Figure 7). The slope is steepest for mode 4 since it has tight loops where it keeps growing lists. Mode 2, the most appropriate for this example, will run through about 2 GB of RAM in 961 seconds. In this time it has explored 240,333 paths to an average depth of 23.8 conditionals. It has executed 1,070,507 C expressions and conditionals. 16 bugs were found, all NULL pointer dereferences.

In interesting effect is that the running rate of *Epitaxis'* interpreter decreases over time (see Figure 6). This is due to increasing size of the data structure that represents memory, and the data structure for the execution tree used to detect paths stepping on themselves. Although memory is held in an interval tree with log2 read and write times, the tree gets large. It may be worth removing older values from the tree if they are completely covered by newer values. Further research needs to be done to determine if older values get completely covered by sibling paths enough to be worth the overhead of finding them and removing them. As the execution tree grows, the algorithm which records code step counts gets slower; it has a deeper tree to search. This additional factor exists in the step count stop modes.

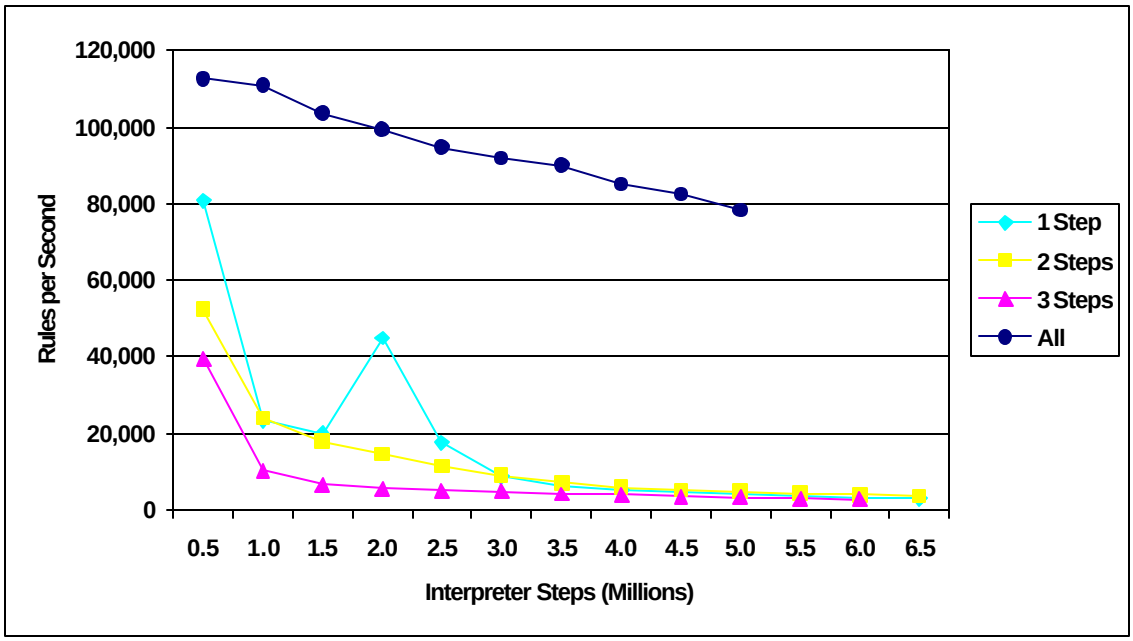

Figure 6 Interpreter Running Rate for retrieve()

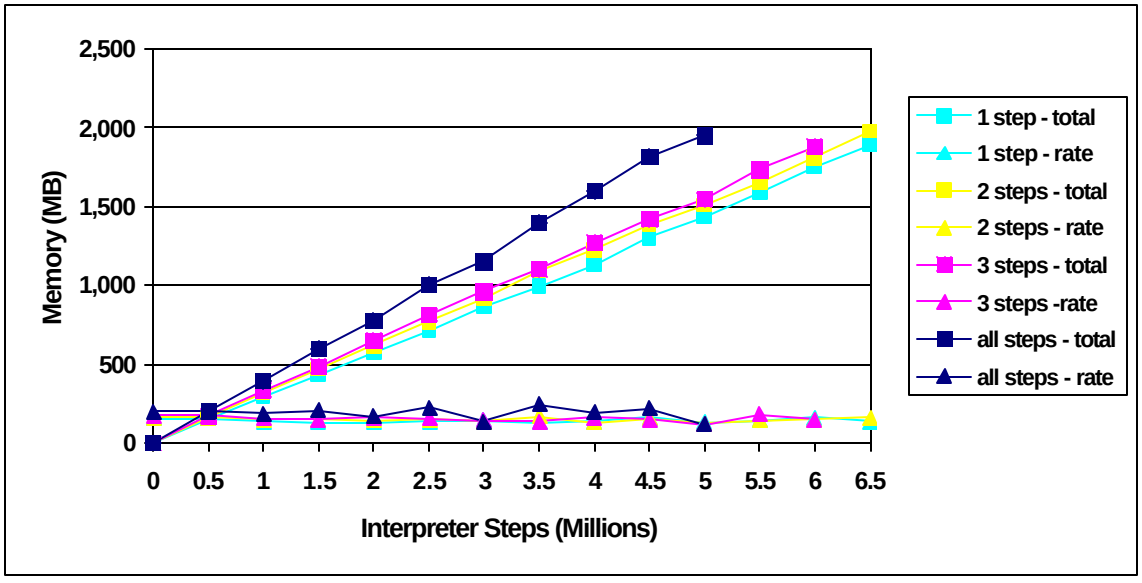

Figure 7 Interpreter Memory Usage for retrieve()

5.3.2 Advanced Semantic Search - Collecting Information

The acme of *Epitaxis'* search is its ability to answer questions through collecting information from throughout the execution space of a program. We present a couple of examples where *Epitaxis* analyzes the use of the `struct` that comprise the elements of the stack from the `retrieve()` example above. The structure of a stack element is shown in Figure 8.

```
struct P_STACK
{
    unsigned is_special   : 1; // special processing on failure
    unsigned is_junction  : 1; // junction point
    unsigned sub_clause   : 1; // Indicates a clause in a junction
    unsigned if_clause    : 1; // Indicates the IF conditional
    unsigned pred_num     : 8; // Built in predicate number

    P_STACK      *prev_stack;   // Link to prior entry
    CONS_CELL    *this_request; // Predicate being RETRIEVEd
    PC_BINDING   *answer;       // Bindings after unification
    PC_BINDING   *bindings;     // Bindings before unification
    P_STACK      *junction;     // previous junction
                                // Points to the junction
                                // Points to the subclause
    union
    {
    CONS_CELL    *back_bindings;    // Undo if fail
    P_STACK      *prev_sub_clause;  // previous sub-clause
    } x;

    union
    {
    FACT         *next_assertion;   // Next predicate to RETRIEVE
    GENERATOR    *next_object;      // Next object to RETRIEVE
    CONS_CELL    *recover_clause;   // Recover on CATCH or HANDLER
    CONS_CELL    *findall_list;     // Accumulated list
    S_EXPRESSION *saved_value;      // Generated value
    } y;
};
```

Figure 8 `struct` definition for stack element

The `pred_num` member field indicates the type of built-in prolog predicate and determines which `union` fields are appropriate. Also the bit fields `is_special`, `is_junction`, `sub_clause`, `if_clause` are set for various types of predicates. These indicators are necessary since C does not support polymorphism and has to be implemented manually. The problem is to determine if all these member fields are being used consistently. In the first

example *Epitaxis* is set to gather all the writes to these five member fields. This information is passed to a *formal concept analysis* algorithm to produce a *concept lattice* representing the equivalence classes of writes to the `struct` elements. The resulting *concept lattice* is shown in Figure 9.

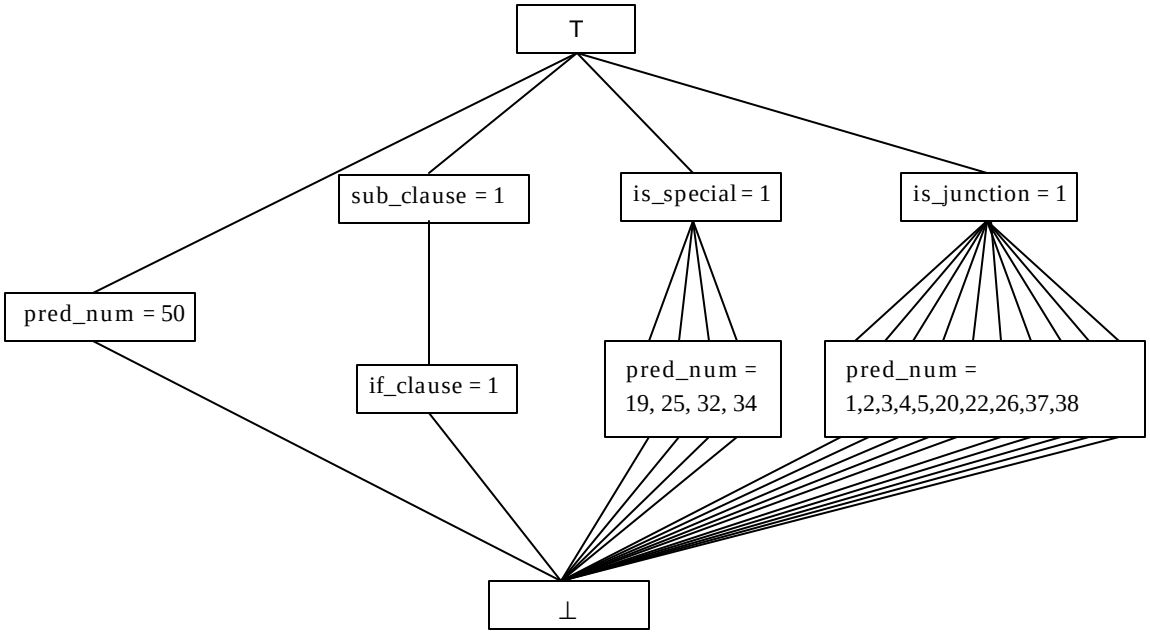

Figure 9: Concept lattice for writes to type fields within a struct

The lattice shows that there is no overlap of predicates between `is_special` and `is_junction`. It also shows that some entries marked `sub_clause` can also be marked `if_clause` and that these are not built-in predicates (as they have no `pred_num` written). Finally it shows that `pred_num`

50 is neither `is_special`, `is_junction`, a `sub_clause` nor within an `if_clause`.

As another example, we use the feature where *Epitaxis* can also look at a member field within a member field. Certain predicates such as `AND`, `OR`, `NOT`, etc. take other predicates as parameters. The `junction` field of the sub clause's `P_STACK` entry (which should have member `sub_clause` = 1) is supposed to point to the containing `AND`, `OR`, `NOT`, etc. predicate. We can test for this by analyzing the `pred_num` member field of the `struct` in the `junction` field. It should contain only predicate numbers of junction type predicates. Figure 10 shows the resulting *concept lattice*. We also check for correct use of `union x`. When a stack element is a junction the `x.prev_sub_clause` member field should be used, otherwise `x.back_bindings` member field should be used.

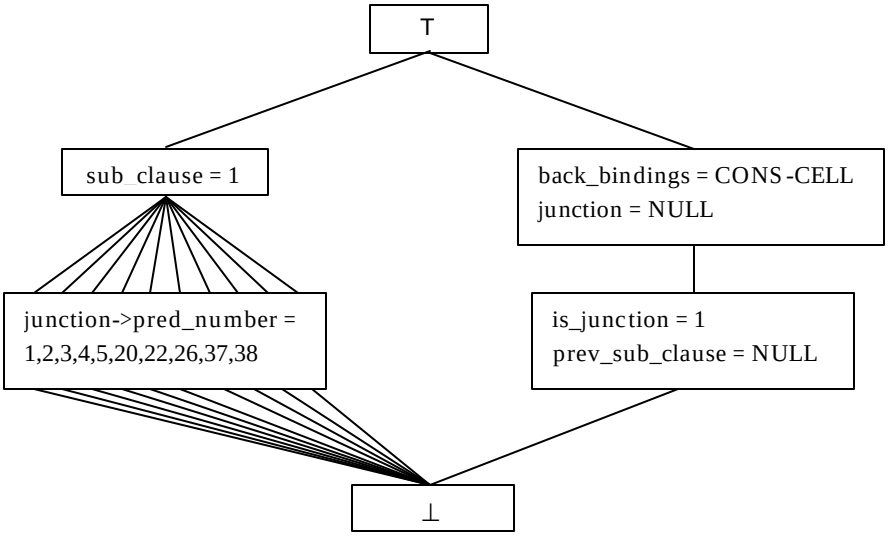

Figure 10 Concept lattice for nested member access and union use

The resulting *concept lattice* shows that all elements marked `sub_clause` (meaning they are a clause within a junction) have a value in their `junction` member field and lists the corresponding `pred_num` of that stack element. All the listed `pred_num` are junction predicates. The other chain shows stack entries that have no value in the junction member field (these are not sub-clauses). A subset of these are marked as junction elements, and have had their `prev_sub_clause` member field initialized to `NULL`. The test was not run far enough for there to be previous sub clauses so only an initialization value is present. Note also that all stack elements created are initialized as non-junctions

and hence the back_bindings member field shows its CONS-CELL value. However, any element subsequently marked as is_junction has that same field (under the union name prev_sub_clause) initialized to NULL. Had the example run further nodes showing values of type P_STACK would be in the junction and prev_sub_clause member fields.

"Good judgment comes from experience. And where does experience come from? Experience comes from bad judgment."
--Mark Twain

"The universe is full of magical things patiently waiting for our wits to grow sharper."
--Eden Phillpots

Chapter 6

6. Conclusion

6.1 Reflections

The goal of *Epitaxis* is twofold. First to implement a modeling language that can be used to build a model[1] of software such that it faithfully represents and exposes those aspects of software a programmer wants to understand. Second to implement a query language that automates extracting that information. It is extremely useful to be able to automate this as a programmer's memory and attention span impose severe limits on his ability to do this

[1] Model of the software itself, not of the application domain using the software

unaided. Any deficiency in the programmer's ability to follow code generally results in the introduction of bugs.

This is an interesting problem for several reasons. Given that software is represented to the human as a string of text characters, most of the information content in it is implicit. Not only is the information implicit, it exists on different levels of abstraction. It is therefore necessary to make several transformations to the original source code to expose the content. Parsing, control flow graph generation, and even execution state space generation (using symbolic execution) are transformations used to generate representations of these levels. These different transformations require different data structures to represent the result as well as different methodologies to search these representations. However much is gained if these levels are integrated.

One of the hallmarks of human intelligence is the ability to abstract and to create further abstract representations of representations to use to thereby leverage the reasoning process. This is one of the key design goals of the *Epitaxis* system. That is, to be able to take entities within the model of the software system, abstract them, represent them, and then use them for a higher (more leveraged) level of reasoning. The actual decision as to the *when* and *what* of these abstractions are currently left to the programmer. What *Epitaxis* is designed to do

is to give the programmer the ability to do this, by providing the machinery to create the abstractions and then query them within the modeling environment.

6.1.1 Modeling

One of the difficulties of finding information or bugs using symbolic execution systems that execute on assembly or machine code is that the executable code, while a transformation of the source code into a more semantic representation, has had a lot of information thrown out. An important thing to remember is that running C code is already a form of abstract interpretation. All addresses of the same location, even if read or written as different types are abstracted to the same value, i.e. the address. The type information is abstracted away. The key to *Epitaxis'* symbolic execution is to keep this information. It has a bearing on correct semantics. It is used to find a broader range of content.

There are many levels of modeling going on here. There is of course what the program is intended to model: the application. There is the source code. There is also the model that the compiler produces, the *CPU model*, which is a translation of the source code into a model that the CPU understands. This model is designed for efficiency; anything unneeded has been removed. This model also has singularities, i.e. states that have no valid continuation (division

by zero, dereferencing of non-existent memory, dereferencing of ill-formatted data, etc.). It is up to the programmer to ensure that these singularities never occur. This is extremely difficult. Ultimately, bugs represent an inconsistency in the application model of the software. However, bugs can express themselves on different levels of the modeling hierarchy. There is a correlation between how soon a bug expresses itself and how low in the modeling hierarchy the bug is. Inconsistencies in the application domain might express as incorrect results and not become noticed for a long time. Some might later express as singularities in the *CPU model*. They eventually become obvious, but may not for a while. The moral is that the higher the modeling level of the inconsistency detected the better.

There are two main ways a model can deviate from what it is modeling. The first way is the simple fact that the model is an *abstraction*, which means that things from reality have been removed from the model. This is done for two reasons: 1) the detail is irrelevant 2) having the detail is too representationally expensive. The second way is that the model might remember history that the reality has no way to hold. This might be called *adstraction* rather than *abstraction*. Certain facts might only become apparent by knowing the objects history that are

not apparent from knowing the objects current state. (Often various histories can reach the same state). In *Epitaxis*, some semantic history is maintained that has no representation in current state of executing C code. An example of this is in conversions between integers and pointers. If something gets casted into an integer the fact that it is the valid address of something and what that something is has no representation within an integer. However this information is crucial if the integer is casted back to an address and dereferenced. Without this history it is not possible to distinguish between an integer that can be safely casted to an address and dereferenced and one that cannot. Until the system crashes, a lot of damage can happen to the structure.

Another example of *adstraction* is remembering "spatial" context. In computing a reference to a member within some structure C only represents the address. The fact that the address is within some `struct` with various properties is lost. However this information can be useful because it implies information about the semantics of values stored nearby and conceptually related.

What makes this interesting from the perspective of this work is that knowing the history of some value can allow a bug to be identified sooner than that bug would manifest in the *CPU model*. A difficulty of debugging is that because history is abstracted away in the *CPU model*, numerous CPU states are

compressions of various correct and non-correct states into one state. So the bug will not manifest until this compressed state collides with some other decision point that differentiates between the histories far removed from the source.

6.1.2 Querying

This research is driven by a real practical need. As programmers write, understand, refactor, and debug code they ask various kinds of questions about the code. This research is about automating the answering of these questions. These questions can broadly be characterized as lexical, syntactic, and semantic based. It is interesting to reflect on how the nature of search changes as the search goes from lexical to syntactic to semantic.

In lexical search there is really no choice of direction. The search starts at the beginning and goes until the object is found. Here the search is looking for an object of some type and with some name. The most sophisticated variant is wild carding on the name and finding a set of objects with a particular pattern to their name. Here the search is about identity. There is no context, which may be what is needed if the query is to understand naming conventions in the software.

In syntactic search the search usually involves finding an entity or set of entities in contextual (or structural) relationship to another entity or set of

entities. It matters where you start and it matters how you choose what direction to take. The search is in large part determined by choosing what direct to take. You still have a finite search space. You are walking a tree and choosing what branch to take based on the type of node you are at (finite state machine) and/or based also on nodes collected before and their relation to the type of node you are at (pushdown automata). Syntactic search is about understanding context.

Things get much more interesting in semantic search; you need to go everywhere. You are searching amongst cause and effect relationships. The cause and effect relations unfold over time and since it is not possible to know what eventually effects what, you are generally forced to explore the entire search space. The search space is usually infinite. Structurally you are walking a graph (the control flow graph), but conceptually you are walking a tree (the execution tree). At the semantic level there is a value space that controls the relationship between the graph and the tree. In general there are four kinds of things you can look for: 1) semantic violations, 2) cause and effect relationships, 3) value relationships, and 4) functional relationships.

Semantic violations are a breakdown of the cause and effect in execution i.e. a point in execution does not have a valid continuation or an inconsistent

data state i.e. a state that system was not designed to model. These can be *CPU model* based, *program model* based, or *application model* based. *CPU model* violations are the most straight forward. These include division by zero, null value dereference, and array index out of bounds. They represent operators that have no result for the set of arguments. They are generally easy to spot since the CPU often throws an exception when one is reached. *Program model* violations are a little more abstract. They are often the bug that eventually leads to an *CPU model* violation. These include bugs in the implementation of a data structure, an improper calculation, or an unanticipated range of input value. *Application model* violations are a mistake in the model that the software implements. These are beyond the scope of this research.

Questions involving cause and effect relationships are about where a value or effect came from. These are useful to understand semantic violations. Generally a programmer works backwards from a symptom (the effect) to the cause.

Questions involving value relationships have to do with relationships between values. (Is the type of the value in field x always y when field z == 3). These are useful to understand the data representations in the model. Valid data

representations typically mandate certain relationships between its parts. Knowing if and where these are violated help to find bugs and to refine the data model.

Questions involving functional relations have to do with the relationship between a function, its arguments, and its result. Functions are defined in code for a number of reasons. From mundane reasons like packaging up often repeated code sequences, to representing programming domain features like getting the value associated with a key, to modeling application domain features like representing the behavior of an agent. A function often models information extraction. The useful idea here is that a function often is the means to answering a query if it could just be executed in the correct context. A future feature for *Epitaxis* is to allow functions (from the program under query) to be called in user specified semantic contexts to answer a question or to extract information to be collected.

The usefulness of *Epitaxis* is in part due to the ephemeral nature of semantic access points. Syntactic structures are static and well defined. Locating features in them or instrumenting them is relatively straight forward. It is much harder to locate or to even find a way to name a point in semantic structures.

These are by nature dynamic and conceptual (as opposed to static and contextual). *Epitaxis* supplies a way to name points in semantic space and either ask a question there or collect information from it (which often only have meaning as an ensemble collected over the execution space).

6.2 Further Research

6.2.1 Further Opening Of The Interpreter

A long standing dichotomy in computer science is between the writer of software and the user of software. Typically the writer defines what the software can do and the user is stuck with it. He might like to change how some small aspect works, but is out of luck. *Open program design* (sometimes called *metaobject protocol*) allows the user to also become author. This is particularly suitable in the present application as the user of this system is a programmer. The idea behind *open program design* is to structure the implementation as an object oriented program so that various aspects of the implementation can be modified by overriding methods that define the behavior of the system.

Epitaxis' symbolic interpreter is *open*. All the operators of the interpreter are defined as methods selected by the types of the arguments. Other functions of the interpreter such as signaling errors, detecting if and how to collect information, binding function parameters, and allocating memory are also implemented as overridable methods. Improvement is needed to make this system much more systematic and complete. Not all the execution points that

should be open have been identified and methodized; these need to be increased. Some of the overridable methods need to be broken down into sub-methods so it is possible to keep most of the functionality and change only part of it. These improvements will make *Epitaxis* more flexible in practice, not just in theory.

6.2.2 Automated Refactoring

Since *Epitaxis* is very good at locating structural relationships and *Epitaxis* has a search methodology that can build and attach structures based on that search, it should be possible to make *Epitaxis* find and change code in a systematic way. A whole library of refactoring searches can be implemented. It will be an interesting research challenge to see how far this refactoring can be pushed into code writing.

6.2.3 Multi-Threaded Support

Epitaxis' symbolic interpreter already has multi-*path* support. It can run multiple paths of the program in "parallel". This support is used internally. It is just a hair's breath away to externalize this machinery such that the software under symbolic execution can be multi-*threaded*. Interpreter operators for fork, join, wait, sleep, resume, etc, need to be added. These would be straight forward

to add; the interpreter would then be able to symbolically execute multi-*threaded* code. The real work would be to add the query ability to ask questions of the multi-*threaded* code.

6.2.4 Object Oriented Language Support

Epitaxis currently supports querying programs in C. Adding the grammar, CFG generation rules, and execution rules for an object oriented language such as C++ would be straight forward. In addition, execution support would have to be added to the interpreter. It needs to be determined what additional query support would need to be added to handle the types of questions that could be asked of object oriented code.

6.2.5 Improving The Constraint Solver

As in every symbolic execution system, the constraint solver is never powerful enough. Improving this is a research field in itself. A more contained improvement would be to add "constrained" proxy-values. This would allow for relationships between different proxy-values such as x is 3 greater than y. This would allow for more accurate determination of *feasible* paths.

6.2.6 Understand The External Environment

Currently, *Epitaxis* has no awareness of the external environment. It has no way to represent an external state that a program might store in the file system. Reading these values back in may constrain execution behavior and reduce false positives. *Epitaxis* also has only a very limited knowledge of standard C library routines. It currently only has representations for malloc, free, longjmp, and setjmp. At a minimum the various string access functions such as memcpy, memset, memchr, etc. functions should have representation since many bugs manifest in these calls. One could always include the library code source but this is cumbersome and many of these routines are in assembly. *Epitaxis* has no ability to handle embedded assembly code. *Epitaxis* also does not understand the semantics of `argc` and `argv`.

6.2.7 Further Forms Of Analysis

The semantic level query of *Epitaxis* is only a starting point. Much more intelligence can be built into where and how to collect information. There needs to be a wider choice of *collect points*, a wider range of filters, and more ways to collect or relate or pieces of linked data structures. It would be useful if functions within the application were also available to be used for assertions or filters. In

addition, other forms of processing the information collected (besides FCA) can be developed.

Appendix A

Rules for the execution of C code:

```
;;;;;;;;;;;;;;;;;;;;;;;;;; Instructions ;;;;;;;;;;;;;;;;;;;;;;;;;;

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::noop-instruction :NEXT ?next))
      (EXECUTE-INSTRUCTION ?next))

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::break-instruction :NEXT ?next))
      (EXECUTE-INSTRUCTION ?next))

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::continue-instruction :NEXT ?next))
      (EXECUTE-INSTRUCTION ?next))

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::goto-instruction :NEXT ?next))
      (EXECUTE-INSTRUCTION ?next))

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::expression-instruction
                                       :ROOT ?root
                                       :LEAVES ?leaves
                                       :NEXT ?next))
      (SEQUENCE (EXECUTE-EXPRESSION ?root ?e-value)
                (EXECUTE-INSTRUCTION ?next)))

(===> (EXECUTE-INSTRUCTION ?(di CLASSES-C::declaration-instruction
                                         :TYPE ?type
                                         :NEXT ?next))
      (SEQUENCE (DECLARE             ?di ?type)
                (EXECUTE-INSTRUCTION ?next)))

(===> (EXECUTE-INSTRUCTION ?(sdi CLASSES-C::scalar-declaration-instruction
                                          :NEXT ?next))
      (SEQUENCE (DECLARE-SCALAR      ?sdi)
                (EXECUTE-INSTRUCTION ?next)))

(===> (EXECUTE-INSTRUCTION ?(pdi CLASSES-C::pointer-declaration-instruction
                                           :NEXT ?next))
      (SEQUENCE (DECLARE-POINTER     ?pdi)
                (EXECUTE-INSTRUCTION ?next)))

(===> (EXECUTE-INSTRUCTION ?(adi CLASSES-C::array-declaration-instruction
                                         :NEXT ?next))
      (SEQUENCE (DECLARE-ARRAY       ?adi)
                (EXECUTE-INSTRUCTION ?next)))

(===> (EXECUTE-INSTRUCTION ?(fdi CLASSES-C::function-declaration-instruction
                                         :NEXT ?next))
      (SEQUENCE (DECLARE-FUNCTION    ?fdi)
                (EXECUTE-INSTRUCTION ?next)))

(===> (EXECUTE-INSTRUCTION ?(rdi CLASSES-C::reference-declaration-instruction
                                         :NEXT ?next))
```

```
      (SEQUENCE (DECLARE-REFERENCE   ?rdi)
                (EXECUTE-INSTRUCTION ?next)))

(===> (EXECUTE-INSTRUCTION ?(bi CLASSES-C::bind-instruction
                                      :SOURCE ?fdo
                                      :NEXT   ?next))
      (SEQUENCE (BIND               ?fdo)
                (EXECUTE-INSTRUCTION ?next)))
(===> (EXECUTE-INSTRUCTION ?(ui CLASSES-C::unbind-instruction
                                      :SOURCE ?fdo
                                      :NEXT   ?next))
      (SEQUENCE (UNBIND             ?fdo)
                (EXECUTE-INSTRUCTION ?next)))

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::return-value-instruction :NEXT ?next))
      (SEQUENCE (RETURN T)
                (EXECUTE-INSTRUCTION ?next)))

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::allocate-instruction
                                     :LABEL ?label
                                     :NEXT  ?next))
      (SEQUENCE (ALLOCATE           ?label)
                (EXECUTE-INSTRUCTION ?next)))

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::deallocate-instruction
                                     :LABEL ?label
                                     :NEXT  ?next))
      (SEQUENCE (DEALLOCATE         ?label)
                (EXECUTE-INSTRUCTION ?next)))

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::catch-instruction
                                     :LABEL ?label
                                     :NEXT  ?next))
      (CATCH ?label ?next))

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::initialize-variable-instructio
                                     :LVALUE ?lvalue
                                     :NEXT ?next))
      (SEQUENCE (EXECUTE::INITIALIZE-VARIABLE ?lvalue $0)
                (EXECUTE-INSTRUCTION ?next)))

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::set-variable-instruction
                                     :LVALUE ?lvalue
                                     :VALUE  ?value
                                     :NEXT   ?next))
      (SEQUENCE (EXECUTE::SET-VALUE ?lvalue ?value)
                (EXECUTE-INSTRUCTION ?next)))

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::if-instruction
                                     :TRUE  ?true
                                     :FALSE ?false))
      (TEST (EXECUTE-INSTRUCTION ?true) (EXECUTE-INSTRUCTION ?false)))

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::switch-instruction
                                     :CASES   ?cases
                                     :DEFAULT ?default))
      (SELECT (EXECUTE-INSTRUCTION ?cases) (EXECUTE-INSTRUCTION ?default)))

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::fork-instruction :CLAUSES ?clauses))
```

```
      (FORK ?clauses))

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::halt-instruction ))
      (HALT))

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::error-instruction :ERROR ?error))
      (ERROR ?error))

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::throw-instruction
                                      :LABEL ?label
                                      :VALUE ?value))
      (THROW ?label ?value))

(===> (EXECUTE-INSTRUCTION #I(CLASSES-C::return-instruction))
      (RETURN))


;;;;;;;;;;;;;;;;;;;;;;;;;; Expressions ;;;;;;;;;;;;;;;;;;;;;;;;;;

(===> (EXECUTE-EXPRESSION-LIST (?expr) ?gather)
      (EXECUTE-EXPRESSION ?expr ?gather))

(===> (EXECUTE-EXPRESSION-LIST (?expr . ?next-expr) ?gather)
      (SEQUENCE (EXECUTE-EXPRESSION ?expr ?gather)
                (EXECUTE-EXPRESSION-LIST ?next-expr ?gather)))


(===> (BIND-PARAMETER-LIST (?formal) (?actual . ?))
      (BIND-PARAMETER ?formal ?actual))

(===> (BIND-PARAMETER-LIST (?formal . ?next-formal) (?actual . ?next-actual))
      (SEQUENCE (BIND-PARAMETER ?formal ?actual)
                (BIND-PARAMETER-LIST ?next-formal ?next-actual)))

(===> (BIND-PARAMETER-LIST (?formal . ?next-formal) NIL))
(===> (BIND-PARAMETER ?(pd CLASSES-C::parameter-declaration
                                    :SPECIFIER ?(ds declaration-specifier
                                                    :TYPE ?type)
                                    :DECLARATOR ?decl) ?a-value)
      (EXECUTE-DECLARATOR ?decl ?type ?var)
      :SIDE-EFFECT (EXECUTE::SET-VALUE ?var ?a-value))

(===> (BIND-PARAMETER ?(var CLASSES-C::variable) ?a-value)
      :SIDE-EFFECT (EXECUTE::SET-VARIABLE-VALUE ?var ?a-value))


(===> (EXECUTE-EXPRESSION ?(il CLASSES-C::initializer-list) ?value)
      :VALUE ?il)

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::if-conditional :COND ?cond-1)
                          ?value-1)
      (EXECUTE-EXPRESSION ?cond-1 ?c-value-1)
      :VALUE ?c-value-1)

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::comma-expression
                                      :LEFT  ?left-2
                                      :RIGHT ?right-2)
                          ?value-2)
      (SEQUENCE (EXECUTE-EXPRESSION ?left-2 ?l-value-2)
```

```
               (EXECUTE-EXPRESSION ?right-2 ?r-value-2))
      :VALUE ?l-value-2)

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::conditional-expression
                                       :COND  ?cond-3
                                       :TRUE  ?true-3
                                       :FALSE ?false-3)
                          ?value-3)
      (IF (EXECUTE-EXPRESSION ?cond-3  ?c-value-3)
          (EXECUTE-EXPRESSION ?true-3  ?t-value-3)
          (EXECUTE-EXPRESSION ?false-3 ?f-value-3))
      :VALUE (EXECUTE::QUESTION-MARK ?c-value-3 ?t-value-3 ?f-value-3))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::assignment
                                       :LVALUE     ?lhs-4
                                       :EXPRESSION ?expr-4)
                          ?value-4)
      (SET (EXECUTE-REFERENCE ?lhs-4   ?location-4)
           (EXECUTE-EXPRESSION ?expr-4 ?e-value-4))
      :VALUE       ?e-value-4
      :SIDE-EFFECT (EXECUTE::SET-VALUE ?location-4 ?e-value-4))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::add-assignment
                                       :LVALUE     ?lhs
                                       :EXPRESSION ?expr)
                          ?value)
      (SET (EXECUTE-REFERENCE ?lhs ?location)
           (EXECUTE-EXPRESSION ?expr ?e-value))
      :VALUE       (EXECUTE::|+| (EXECUTE::GET-VALUE ?location) ?e-value)
      :SIDE-EFFECT (EXECUTE::SET-VALUE ?location ?value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::sub-assignment
                                       :LVALUE     ?lhs
                                       :EXPRESSION ?expr)
                          ?value)
      (SET (EXECUTE-REFERENCE  ?lhs  ?location)
           (EXECUTE-EXPRESSION ?expr ?e-value))
      :VALUE       (EXECUTE::|-| (EXECUTE::GET-VALUE ?location) ?e-value)
      :SIDE-EFFECT (EXECUTE::SET-VALUE ?location ?value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::mul-assignment
                                       :LVALUE     ?lhs
                                       :EXPRESSION ?expr) ?value)
      (SET (EXECUTE-REFERENCE ?lhs ?location)
           (EXECUTE-EXPRESSION ?expr ?e-value))
      :VALUE       (EXECUTE::|*| (EXECUTE::GET-VALUE ?location) ?e-value)
      :SIDE-EFFECT (EXECUTE::SET-VALUE ?location ?value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::div-assignment
                                       :LVALUE     ?lhs
                                       :EXPRESSION ?expr)
                          ?value)
      (SET (EXECUTE-REFERENCE  ?lhs  ?location)
           (EXECUTE-EXPRESSION ?expr ?e-value))
      :VALUE       (EXECUTE::|/| (EXECUTE::GET-VALUE ?location) ?e-value)
      :SIDE-EFFECT (EXECUTE::SET-VALUE ?location ?value))
```

```
(===> (EXECUTE-EXPRESSION #I(CLASSES-C::mod-assignment
                                        :LVALUE     ?lhs
                                        :EXPRESSION ?expr)
                          ?value)
      (SET (EXECUTE-REFERENCE  ?lhs  ?location)
           (EXECUTE-EXPRESSION ?expr ?e-value))
      :VALUE       (EXECUTE::|%| (EXECUTE::GET-VALUE ?location) ?e-value)
      :SIDE-EFFECT (EXECUTE::SET-VALUE ?location ?value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::lsh-assignment
                                        :LVALUE     ?lhs
                                        :EXPRESSION ?expr)
                          ?value)
      (SET (EXECUTE-REFERENCE  ?lhs  ?location)
           (EXECUTE-EXPRESSION ?expr ?e-value))
      :VALUE       (EXECUTE::|<<| (EXECUTE::GET-VALUE ?location) ?e-value)
      :SIDE-EFFECT (EXECUTE::SET-VALUE ?location ?value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::rsh-assignment
                                        :LVALUE     ?lhs
                                        :EXPRESSION ?expr)
                          ?value)
      (SET (EXECUTE-REFERENCE  ?lhs  ?location)
           (EXECUTE-EXPRESSION ?expr ?e-value))
      :VALUE       (EXECUTE::|>>| (EXECUTE::GET-VALUE ?location) ?e-value)
      :SIDE-EFFECT (EXECUTE::SET-VALUE ?location ?value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::and-assignment
                                        :LVALUE     ?lhs
                                        :EXPRESSION ?expr)
                          ?value)
      (SET (EXECUTE-REFERENCE  ?lhs  ?location)
           (EXECUTE-EXPRESSION ?expr ?e-value))
      :VALUE       (EXECUTE::|&| (EXECUTE::GET-VALUE ?location) ?e-value)
      :SIDE-EFFECT (EXECUTE::SET-VALUE ?location ?value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::eor-assignment
                                        :LVALUE     ?lhs
                                        :EXPRESSION ?expr)
                          ?value)
      (SET (EXECUTE-REFERENCE  ?lhs  ?location)
           (EXECUTE-EXPRESSION ?expr ?e-value))
      :VALUE       (EXECUTE::|^| (EXECUTE::GET-VALUE ?location) ?e-value)
      :SIDE-EFFECT (EXECUTE::SET-VALUE ?location ?value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::or-assignment
                                        :LVALUE     ?lhs
                                        :EXPRESSION ?expr)
                          ?value)
      (SET (EXECUTE-REFERENCE  ?lhs  ?location)
           (EXECUTE-EXPRESSION ?expr ?e-value))
      :VALUE       (EXECUTE::\| (EXECUTE::GET-VALUE ?location) ?e-value)
      :SIDE-EFFECT (EXECUTE::SET-VALUE ?location ?value))


(===> (EXECUTE-EXPRESSION #I(CLASSES-C::logical-or-expression
                                        :LEFT  ?left
                                        :RIGHT ?right)
                           ?value)
```

```
      (AND (EXECUTE-EXPRESSION ?left ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value))
      :VALUE (EXECUTE::\|\| ?l-value ?r-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::logical-and-expression
                                       :LEFT  ?left
                                       :RIGHT ?right)
                          ?value)
      (AND (EXECUTE-EXPRESSION ?left  ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value))
      :VALUE (EXECUTE::|&&| ?l-value ?r-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::bit-or-expression
                                       :LEFT  ?left
                                       :RIGHT ?right)
                          ?value)
      (AND (EXECUTE-EXPRESSION ?left  ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value))
      :VALUE (EXECUTE::\| ?l-value ?r-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::bit-eor-expression
                                       :LEFT  ?left
                                       :RIGHT ?right)
                          ?value)
      (AND (EXECUTE-EXPRESSION ?left  ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value))
      :VALUE (EXECUTE::|^| ?l-value ?r-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::bit-and-expression
                                       :LEFT  ?left
                                       :RIGHT ?right)
                          ?value)
      (AND (EXECUTE-EXPRESSION ?left ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value))
      :VALUE (EXECUTE::|&| ?l-value ?r-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::eq-expression
                                      :LEFT  ?left
                                      :RIGHT ?right)
                           ?value)
      (AND (EXECUTE-EXPRESSION ?left  ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value)
      :VALUE (EXECUTE::|==| ?l-value ?r-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::neq-expression
                                       :LEFT ? left
                                       :RIGHT ?right)
                          ?value)
      (AND (EXECUTE-EXPRESSION ?left  ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value))
      :VALUE (EXECUTE::|!=| ?l-value ?r-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::lt-expression
                                       :LEFT  ?left
                                       :RIGHT ?right)
                          ?value)
      (AND (EXECUTE-EXPRESSION ?left ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value))
      :VALUE (EXECUTE::|<| ?l-value ?r-value))
```

```
(===> (EXECUTE-EXPRESSION #I(CLASSES-C::le-expression
                                        :LEFT  ?left
                                        :RIGHT ?right)
                          ?value)
      (AND (EXECUTE-EXPRESSION ?left  ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value))
      :VALUE (EXECUTE::|<=| ?l-value ?r-value) )

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::gt-expression
                                        :LEFT  ?left
                                        :RIGHT ?right)
                          ?value)
      (AND (EXECUTE-EXPRESSION ?left  ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value))
      :VALUE (EXECUTE::|>| ?l-value ?r-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::ge-expression
                                        :LEFT  ?left
                                        :RIGHT ?right)
                          ?value)
      (AND (EXECUTE-EXPRESSION ?left  ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value))
      :VALUE (EXECUTE::|>=| ?l-value ?r-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::rsh-expression
                                        :LEFT  ?left
                                        :RIGHT ?right)
                          ?value)
      (AND (EXECUTE-EXPRESSION ?left  ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value))
      :VALUE (EXECUTE::|>>| ?l-value ?r-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::lsh-expression
                                        :LEFT  ?left
                                        :RIGHT ?right)
                          ?value)
      (AND (EXECUTE-EXPRESSION ?left  ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value))
      :VALUE (EXECUTE::|<<| ?l-value ?r-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::add-expression
                                        :LEFT  ?left
                                        :RIGHT ?right)
                          ?value)
      (AND (EXECUTE-EXPRESSION ?left  ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value))
      :VALUE (EXECUTE::|+| ?l-value ?r-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::sub-expression
                                        :LEFT  ?left
                                        :RIGHT ?right)
                          ?value)
      (AND (EXECUTE-EXPRESSION ?left  ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value))
      :VALUE (EXECUTE::|-| ?l-value ?r-value))
```

```
(===> (EXECUTE-EXPRESSION #I(CLASSES-C::mul-expression
                                      :LEFT  ?left
                                      :RIGHT ?right)
                          ?value)
      (AND (EXECUTE-EXPRESSION ?left  ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value))
      :VALUE (EXECUTE::|*| ?l-value ?r-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::div-expression
                                      :LEFT  ?left
                                      :RIGHT ?right)
                          ?value)
      (AND (EXECUTE-EXPRESSION ?left  ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value))
      :VALUE (EXECUTE::|/| ?l-value ?r-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::mod-expression
                                      :LEFT  ?left
                                      :RIGHT ?right)
                          ?value)
      (AND (EXECUTE-EXPRESSION ?left  ?l-value)
           (EXECUTE-EXPRESSION ?right ?r-value))
      :VALUE (EXECUTE::|%| ?l-value ?r-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::not-expression :ARG ?arg) ?value)
      (EXECUTE-EXPRESSION ?arg ?a-value)
      :VALUE (EXECUTE::|!| ?a-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::complement-expression :ARG ?arg)
                          ?value)
      (EXECUTE-EXPRESSION ?arg ?a-value)
      :VALUE (EXECUTE::|~| ?a-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::negate-expression :ARG ?arg) ?value)
      (EXECUTE-EXPRESSION ?arg ?a-value)
      :VALUE (EXECUTE::NEGATE ?a-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::plus-expression :ARG ?arg) ?value)
      (EXECUTE-EXPRESSION ?arg ?a-value)
      :VALUE ?a-value)

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::pre-incr-expression :ARG ?lhs) ?value)
      (SET (EXECUTE-REFERENCE ?lhs ?location))
      :VALUE       (EXECUTE::|+| (EXECUTE::GET-VALUE ?location) 1)
      :SIDE-EFFECT (EXECUTE::SET-VALUE ?location ?value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::pre-decr-expression :ARG ?lhs) ?value)
      (SET (EXECUTE-REFERENCE ?lhs ?location))
      :VALUE       (EXECUTE::|-| (EXECUTE::GET-VALUE ?location) 1)
      :SIDE-EFFECT (EXECUTE::SET-VALUE ?location ?value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::post-incr-expression :ARG ?lhs) ?value)
      (SET (EXECUTE-REFERENCE ?lhs ?location))
      :VALUE       (EXECUTE::GET-VALUE ?location)
      :SIDE-EFFECT (EXECUTE::SET-VALUE ?location (EXECUTE::|+| ?value 1)))
```

```
(===> (EXECUTE-EXPRESSION #I(CLASSES-C::post-decr-expression :ARG ?lhs) ?value)
      (SET (EXECUTE-REFERENCE ?lhs ?location))
      :VALUE       (EXECUTE::GET-VALUE ?location)
      :SIDE-EFFECT (EXECUTE::SET-VALUE ?location (EXECUTE::|-| ?value 1)))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::sizeof-expression :ARG ?arg) ?value)
      (EXECUTE-EXPRESSION ?arg ?a-value)
      :VALUE (EXECUTE::SIZEOF ?a-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::cast-expression
                                       :CAST       ?cast
                                       :EXPRESSION ?expr)
                          ?value)
      (EXECUTE-EXPRESSION ?expr ?e-value)
      :VALUE (EXECUTE::CAST ?e-value ?cast))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::addr-of-expression
                                       :ARG ?(fn CLASSES-C::function-name))
                          ?value)
      :VALUE (EXECUTE::GET-FUNCTION-ADDRESS ?fn))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::addr-of-expression :ARG ?arg) ?value)
      (EXECUTE-REFERENCE ?arg ?a-value)
      :VALUE (EXECUTE:ADDRESS-OF ?a-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::pointer-expression :ARG ?arg) ?value)
      (EXECUTE-EXPRESSION ?arg ?a-value)
      :VALUE (EXECUTE::INDIRECT ?a-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::pointer-access-expression
                                       :TAG    ?tag
                                       :MEMBER ?member)
                          ?value)
      (EXECUTE-EXPRESSION ?tag ?t-value)
      :VALUE (EXECUTE::GET-MEMBER-INDIRECT-VALUE ?t-value ?member))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::direct-access-expression
                                       :TAG    ?tag
                                       :MEMBER ?member)
                          ?value)
      (EXECUTE-EXPRESSION ?tag ?t-value)
      :VALUE (EXECUTE::GET-MEMBER-VALUE ?t-value ?member))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::array-expression
                                       :ARRAY ?array
                                       :INDEX ?index)
                          ?value)
      (AND (EXECUTE-EXPRESSION ?array ?a-value)
           (EXECUTE-EXPRESSION ?index ?i-value))
      :VALUE (EXECUTE::GET-ARRAY-VALUE ?a-value ?i-value))

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::paren-expression :EXPRESSION ?expr)
                          ?value)
      (EXECUTE-EXPRESSION ?expr ?e-value)
      :VALUE ?e-value)
```

```
(===> (EXECUTE-EXPRESSION #I(CLASSES-C::function-expression
                                    :FUN  ?fun
                                    :ARGS NIL)
                          ?value)
      (SEQUENCE (EXECUTE-EXPRESSION ?fun ?f-value)
                (CALL ?c-value ?f-value))
      :VALUE ?c-value)

(===> (EXECUTE-EXPRESSION #I(CLASSES-C::function-expression
                                    :FUN  ?fun
                                    :ARGS ?actuals)
                          ?value)
      (SEQUENCE (EXECUTE-EXPRESSION ?fun ?f-value)
                (COLLECT ?a-value :DIRECTION :QUEUE)
                (EXECUTE-EXPRESSION-LIST ?actuals ?a-value)
                (IS ?a-values (RESULT-OF ?a-value))
                (CALL ?c-value ?f-value ?a-values))
      :VALUE ?c-value)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Primarys ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(===> (EXECUTE-EXPRESSION ?(var variable) ?value)
      :VALUE (EXECUTE::GET-VARIABLE-VALUE ?var))

(===> (EXECUTE-EXPRESSION ?(fp CLASSES-C::formal-parameter) ?value)
      :VALUE (EXECUTE::GET-VARIABLE-VALUE ?fp))

(===> (EXECUTE-EXPRESSION ?(uv CLASSES-C::undeclared-variable) ?value)
      :VALUE (make-instance CLASSES-C::unspecified-value))

(===> (EXECUTE-EXPRESSION ?(mn CLASSES-C::member-name) ?value)
      :VALUE ?mn)

(===> (EXECUTE-EXPRESSION ?(fn CLASSES-C::function-name) ?value)
      :VALUE (EXECUTE::GET-FUNCTION-ADDRESS ?fn))

(===> (EXECUTE-EXPRESSION ?(ufn CLASSES-C::undeclared-function-name) ?value)
      :VALUE (make-instance CLASSES-C::unspecified-value))

(===> (EXECUTE-EXPRESSION ?(tn CLASSES-C::tag-name) ?value)
      :VALUE ?tn)

(===> (EXECUTE-EXPRESSION ?(tn CLASSES-C::type-name) ?value)
      :VALUE ?tn)

(===> (EXECUTE-EXPRESSION ?(tn CLASSES-C::typedef-name) ?value)
      :VALUE ?tn)

(===> (EXECUTE-EXPRESSION ?(ec CLASSES-C::enum-constant) ?value)
      :VALUE (EXECUTE::ENUM-CONSTANT ?ec))

(===> (EXECUTE-EXPRESSION ?(fc CLASSES-C::float-constant :VALUE ?c-value)
                          ?value)
      :VALUE (EXECUTE::CONSTANT ?c-value))

(===> (EXECUTE-EXPRESSION ?(dc CLASSES-C::double-constant :VALUE ?c-value)
                          ?value)
      :VALUE (EXECUTE::CONSTANT ?c-value))
```

```
(===> (EXECUTE-EXPRESSION ?(ldc CLASSES-C::long-double-constant
                                          :VALUE ?c-value)
                          ?value)
      :VALUE (EXECUTE::CONSTANT ?c-value))

(===> (EXECUTE-EXPRESSION ?(ic CLASSES-C::integer-constant :VALUE ?c-value)
                          ?value)
      :VALUE (EXECUTE::CONSTANT ?c-value))

(===> (EXECUTE-EXPRESSION ?(uic CLASSES-C::unsigned-integer-constant
                                          :VALUE ?c-value)
                          ?value)
      :VALUE (EXECUTE::CONSTANT ?c-value))

(===> (EXECUTE-EXPRESSION ?(lic CLASSES-C::long-integer-constant
                                          :VALUE ?c-value)
                          ?value)
      :VALUE (EXECUTE::CONSTANT ?c-value))

(===> (EXECUTE-EXPRESSION ?(ulic CLASSES-C::unsigned-long-integer-constant
                                           :VALUE ?c-value)
                          ?value)
      :VALUE (EXECUTE::CONSTANT ?c-value))

(===> (EXECUTE-EXPRESSION ?(cc CLASSES-C::character-constant :VALUE ?c-value)
                          ?value)
      :VALUE (EXECUTE::CONSTANT ?c-value))

(===> (EXECUTE-EXPRESSION ?(lcc CLASSES-C::long-character-constant
                                          :VALUE ?c-value)
                          ?value)
      :VALUE (EXECUTE::CONSTANT ?c-value))

(===> (EXECUTE-EXPRESSION ?(sc CLASSES-C::string-constant :NAME ?c-value)
                          ?value)
      :VALUE (EXECUTE::CONSTANT ?c-value))

(===> (EXECUTE-EXPRESSION ?(lsc CLASSES-C::long-string-constant :NAME ?c-value)
                          ?value)
      :VALUE (EXECUTE::CONSTANT ?c-value))


;;;;;;;;;;;;;;;;;;;;;; References for LHS writes ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(===> (EXECUTE-REFERENCE ?(var CLASSES-C::variable) ?value)
      :VALUE (EXECUTE::GET-VARIABLE-ADDRESS ?var))

(===> (EXECUTE-REFERENCE #I(CLASSES-C::array-expression
                                       :ARRAY ?array
                                       :INDEX ?index)
                         ?value)
      (SEQUENCE (EXECUTE-EXPRESSION ?array ?a-value)
                (EXECUTE-EXPRESSION ?index ?i-value))
      :VALUE (EXECUTE::GET-ARRAY-ADDRESS ?a-value ?i-value) )
```

```
(===> (EXECUTE-REFERENCE #I(CLASSES-C::direct-access-expression
                                    :TAG    ?tag
                                    :MEMBER ?member)
                         ?value)
      (EXECUTE-EXPRESSION ?tag ?t-value)
      :VALUE (EXECUTE::GET-MEMBER-ADDRESS ?t-value ?member))

(===> (EXECUTE-REFERENCE #I(CLASSES-C::pointer-access-expression
                                    :TAG    ?tag
                                    :MEMBER ?member)
                         ?value)
      (EXECUTE-EXPRESSION ?tag ?t-value)
      :VALUE (EXECUTE::GET-MEMBER-INDIRECT-ADDRESS ?t-value ?member))

(===> (EXECUTE-REFERENCE #I(CLASSES-C:pointer-expression :ARG ?arg) ?value)
      (EXECUTE-EXPRESSION ?arg ?a-value)
      :VALUE (EXECUTE::GET-INDIRECT-ADDRESS ?a-value))

(===> (EXECUTE-REFERENCE #I(CLASSES-C::cast-expression
                                    :CAST       ?cast
                                    :EXPRESSION ?expr)
                         ?value)
      (EXECUTE-REFERENCE ?expr ?e-value)
      :VALUE (EXECUTE::CAST ?e-value ?cast))

(===> (EXECUTE-REFERENCE #I(CLASSES-C::add-expression
                                    :LEFT  ?left
                                    :RIGHT ?right)
                         ?value)
      (AND (EXECUTE-REFERENCE ?left  ?l-value)
           (EXECUTE-REFERENCE ?right ?r-value))
      :VALUE (EXECUTE::|+| ?l-value ?r-value))

(===> (EXECUTE-REFERENCE #I(CLASSES-C::sub-expression
                                    :LEFT  ?left
                                    :RIGHT ?right)
                         ?value)
      (AND (EXECUTE-REFERENCE ?left  ?l-value)
           (EXECUTE-REFERENCE ?right ?r-value))
      :VALUE (EXECUTE::|-| ?l-value ?r-value))

(===> (EXECUTE-REFERENCE #I(CLASSES-C::paren-expression :EXPRESSION ?expr)
                         ?value)
      (EXECUTE-REFERENCE ?expr ?e-value)
      :VALUE       ?e-value)

(===> (EXECUTE-REFERENCE ?(expr CLASSES-C::expression) ?value)
      (EXECUTE-EXPRESSION ?expr ?value)
      :VALUE ?value)
```

"A couple of months in the laboratory can frequently save a couple of hours in the library."

--Frank Westheimer

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman, "Propagation of Constraints," in *The Structure and Interpretation of Computer Programs*: MIT Press, pp. 542, 1985.

[2] A. V. Aho and S. C. Johnson, "LR Parsing," *Computing Surveys*, vol. 6, no. 2, pp. 99-124, 1974.

[3] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, *The AWK Programmin Language*: Addison-Wesley, 1988.

[4] Alexander Aiken and Brian R. Murphy, "Implementing Regular Tree Expressions," in *FPCA 1991*, 1991, pp. 427-447.

[5] Vladimir Alexiev, "A (Not Very Much) Annotated Bibliography on Integrating Object-Oriented and Logic Programming," ftp://menaik.cs.ualberta.ca/pub/oolog/oolog-bib.ps

[6] James Ambras and Vicki O'Day, "Microscope: A Knowledge-Based Programming Environment," *IEEE Software*, vol. 5, no. 3, pp. 50-58, May, 1988.

[7] Paul Anderson, Thomas Reps, and Tim Teitelbaum, "Design and Implementation of a Fine-Grained Software Inspection Tool," *IEEE Transactions on Software Engineering*, vol. 29, no. 8, pp. 721-733, August, 2003.

[8] Paul Anderson and Tim Teitelbaum, "Software Inspection Using CodeSurfer," presented at Proceedings of the First Workshop on Inspection in Software Engineering 2001, Paris, pp. 1-9.

[9] Cyrille Artho and Armin Biere, "Combined Static and Dynamic Analysis," *Electr. Notes Theor. Comput. Sci.*, vol. 131, pp. 3-14, 2005.

[10] Cyrille Artho, Viktor Schuppan, Armin Biere, Pascal Eugster, Marcel Baur, and Boris Zweimüller, "JNuke: Efficient Dynamic Analysis for Java.," in *CAV*, 2004, pp. 462-465.

[11] Patrizia Asirelli, Pierpaolo Degano, Giorgio Levi, Alberto Martelli, Ugo Montanari, Giuliano Pacini, Franco Sirovich, and Franco Turini, "A Flexible Environment for Program Development Based on a Symbolic Interpreter," *ICSE*, pp. 251-264, 1979.

[12] Llya Bagrak and Olin Shivers, "trx: Regular-Tree Expressions, Now in Scheme," in *Fifth Workshop on Scheme and Functional Programming*, 2004, pp. 21-32.

[13] Thomas Ball, "The Concept of Dynamic Analysis," *ESEC / SIGSOFT FSE*, pp. 216-234, 1999.

[14] Francois Bancilhon, "A logic-programming/object-oriented cocktail," *ACM SIGMOD Record*, vol. 15, no. 3, pp. 11-21, September 1986, 1986.

[15] Bell Canada Inc., "DATRIX Abstract Semantic Graph Reference Manual ", 2000.

[16] David Binkley, "Souce Code Analysis: A Road Map," in *FOSE 2007*: IEEE, 2007.

[17] Barry W. Boehm, "Software Engineering," *IEEE Transactions on Computers*, vol. 25, no. 12, pp. 1226-1241, December, 1976.

[18] Patrice Boizumault, *The Implementation Of Prolog*: Princeton University Press, 1993.

[19] Robert Bowdidge and William Griswold, "Automated Support for Encapsulating Abstract Data Types," in *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1994, pp. 97-110.

[20] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt, "SELECT—a formal system for testing and debugging programs by symbolic execution," presented at Proceedings of the international conference on Reliable software, Los Angeles, California, pp. 234 - 245.

[21] Marc M. Brandis and Hanspeter Mössenböck, "Single-Pass Generation of Static Single-Assignment Form for Structured Languages," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1684-1698, 1994.

[22] Johan Brichau, Coen De Roover, and Kim Mens, "Open Unification for Program Query Languages," in *XXVI International Conference of the Chilean Society of Computer Science (SCCC'07)*. Iquique, Chile, 2007.

[23] Frederick P. Brooks, "No Silver Bullet," *Computer*, vol. 20, no. 4, pp. 10-19, April, 1987.

[24] Bernd Bruegge, Tim Gottschalk, and Bin Luo, "A Framework for Dynamic Program Analyzers," in *Proceedings of the Eighth Annual Conference on Object-*

Oriented Programming Systems, Languages, and Applications. Washington, D.C., United States: ACM Press, 1993, pp. 65-82.

[25] R.I. Bull, A. Trevors, A.J. Malton, and M.W. Godfrey, "Semantic Grep: Regular Expressions + Relational Abstraction," presented at Ninth Working Conference on Reverse Engineering (WCRE'02).

[26] S. Burson, G.B. Kotik, and L.Z. Markosian, "A program transformation approach to automating software re-engineering," presented at Proceedings of the Fourteenth Annual International Computer Software and Applications Conference, 1990. COMPSAC 90., pp. 314 - 322.

[27] William R. Bush, Jonathan D. Pincus, and David J. Sielaff, "A Static Analyzer for Finding Dynamic Programming Errors," *Software Practice and Experience*, vol. 30, pp. 775-802, 2000.

[28] Cristian Cadar, Daniel Dunbar, and Dawson Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," presented at 8th USENIX Symposium on Operating Systems Design and Implementation, San Diego, California, USA, pp. 209-224, December 8-10, 2008.

[29] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson Engler, "EXE: automatically generating inputs of death," in *Proceedings of the 13th ACM conference on Computer and communications security*. Alexandria, Virginia, USA: ACM, 2006.

[30] J.A. Campbell and S. Hardy, "Should Prolog be list or record oriented?," in *Implementations of PROLOG*, J.A. Campball, Ed.: Ellis Horwood, 1984.

[31] Stefano Ceri, Georg Gottlob, and Letizia Tanca, "What You Always Wanted to Know About Datalog (And Never Dared to Ask)," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 1, pp. 146-166, March 1989, 1989.

[32] Pui-Shan Chan and Malcolm Munro, "PUI: A Tool to Support Program Understanding," in *Proceedings of the IEEE 5th International Workshop on Program Comprehension*, 1997, pp. 192-198.

[33] Nigel P. Chapman, *LR Parsing: Theory and Practice*: Cambridge University Press, 1987.

[34] Peter Pin-Shan Chen, "The Entity-Relationship Model – Toward a Unified View of Data," *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9-36, March, 1976.

[35] Weidong Chen and David Scott Warren, "Tabled Evaluation With Delaying for General Logic Programs," *Journal of the Association for Computing Machinery*, vol. 43, no. 1, pp. 20-74, January 1996, 1996.

[36] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy, "The C Information Abstraction System," *IEEE Transactions on Software Engineering*, vol. 16, no. 3, March, 1990.

[37] Lori A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 215-222, September, 1976, 1976.

[38] Lori A. Clarke and Debra J. Richardson, "Applications of Symbolic Evaluation," *Journal of Systems and Software*, vol. 5, no. 1, pp. 15-35, 1985.

[39] Tal Cohen, Joseph Gil, and Itay Maman, "JTL - the Java Tools Language," in *OOPSLA '06*. Portland, Oregon: ACM, 2006.

[40] Patrick Cousot, "Abstract Interpretation Based Formal Methods and Future Challenges," *Informatics*, pp. 183-156, 2001.

[41] Patrick Cousot and Radhia Cousot, "Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proc. 4th ACM Symposium on Principles of Programming Languages*, 1977, pp. 238-252.

[42] David Coward and Darrel Ince, *The Symbolic Execution of Software: The SYMBOL system*: Chapman & Hall, 1995.

[43] P. David Coward, "Symbolic Execution Systems – a review," *Software Engineering Journal November*, pp. 229-239, 1988.

[44] Roger F. Crew, "ASTLOG: A Language for Examining Abstract Syntax Trees," in *Proc. USENIX Conf. Domain-Specific Languages*, 1997.

[45] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451-490, 1991.

[46] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "An Efficient Method of Computing Static Single Assignment Form," presented at POPL 1989, pp. 25-35.

[47] Manuvir Das, "Unification-Based Pointer Analysis with Directional Assignments," *ACM SIGPLAN Notices*, vol. 35, no. 4, pp. 35-46, 2000.

[48] B.A. Davey and H.A. Priestley, *Introduction to Lattices and Order*, 2nd Edition ed: Cambridge University Press, 2002.

[49] Kris De Volder, *Type-Oriented Logic Meta Programming*. Programming Technology Department, Vrije Universiteit Brussel, 1998.

[50] Kris De Volder, "JQuery: A Generic Code Browser with a Declarative Configuration Language," presented at PADL 2006, pp. 88-102.

[51] P. Deransart, A. Ed-Dbali, and L. Cervoni, *Prolog: The Standard*: Springer, 1996.

[52] Franklin L. DeRemer, *Practical Translators for LR(k) Languages*. Dep. Electrical Engineering, MIT, 1969.

[53] Franklin L. DeRemer, "Simple LR(k) grammars," *Communications of the ACM*, vol. 14, no. 7, pp. 453 - 460, July, 1971.

[54] Franklin L. DeRemer and Thomas Pennello, "Efficient Computation of LALR(1) Look-Ahead Sets," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 4, pp. 615-649, October, 1982.

[55] Francoise Detienne, *Software Design – Cognitive Aspects*: Springer, 2001.

[56] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe, "Extended Static Checking," COMPAQ December 18, 1998 1998.

[57] Premkumar T. Devanbu, "GENOA—a customizable, front-end-retargetable source code analysis framework," *ACM Transactions on Software Engineering and Methodology*, vol. 8, no. 2, pp. 177-212, 1999.

[58] Edsger W. Djikstra, "The Humble Programmer," *Communications of the ACM*, vol. 15, no. 10, 1972.

[59] Jurgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter, "GUPRO Generic Understanding of Programs: An Overview," June, 2002.

[60] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon, "Aiding Program Comprehension by Static and Dynamic Feature Analysis," in *Proceedings of the International Conference on Software Maintenance, ICSM '2001*, 2001.

[61] David Evans, "Static Detection of Dynamic Memory Errors," *PLDI*, pp. 44-53, 1996.

[62] David Evans, John Guttag, James Horning, and Yang Meng Tan, "LCLint: A Tool for Using Specifications to Check Code," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*: ACM, 1994.

[63] Cormac Flanagan, K. Rustan, M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata, "Extended Static Checking for Java.," *PLDI*, pp. 234-245, 2002.

[64] Margaret Ann Francel and Spencer Rugaber, "The Relationship of Slicing and Debugging to Program Understanding," *IWPC*, pp. 106-113, 1999.

[65] Koichi Fukunaga and Shin'ichi Hirose, "An Experience with a Prolog-based Object-Oriented Language," in *OOPSLA 1986*: ACM, 1986.

[66] Koichi Fukunaga and Shin-ichi Hirose, "An Experience with a Prolog-based Object-Oriented Language," *OOPSLA 1986*, pp. 224-231, 1986.

[67] Bernhard Ganter and Rudolf Wille, *Formal Concept Analysis: Mathematical Foundations*: Springer Verlag, 1999.

[68] Paul C. Gilmore, "A program for the production from axioms, of proofs for theorems derivable within the first order predicate calculus.," presented at Proceedings of the IFIP Congress (1959), pp. 265-273.

[69] Matthew L. Ginsberg, "Multivalued logics: a uniform approach to reasoning in artificial intelligence," *Computational Intelligence*, vol. 4, pp. 265-316, 1988.

[70] Matthew L. Ginsberg, "Bilattices and modal operators," presented at Proceedings of the 3rd conference on Theoretical aspects of reasoning about knowledge, Pacific Grove, California pp. 273-287.

[71] Patrice Godefroid, Nils Klarlund, and Koushik Sen, "DART: Directed Automated Random Testing," in *PLDI 2005*. Chicago, Illnois, USA: ACM, 2005.

[72] W.G. Griswold, D.C. Atkinson, and C. McCurdy, "Fast, flexible syntactic pattern matching and processing," presented at Proceedings of the Fourth Workshop on Program Comprehension, 1996, Berlin, Germany, pp. 144-153.

[73] Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst, "Dynamic Inference of Abstract Types," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. Portland, ME, USA, 2006, pp. 255-265.

[74] Elnar Hajiyev, *CodeQuest - Source Code Querying with Datalog*. Computing Laboratory, Oxford University, 2005.

[75] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor, "CodeQuest: Scalable Source Code Queries with DataLog," in *Procs. of the European Conference on Object-Oriented Programming (ECOOP)*, 2006.

[76] M.T. Harandi and J.Q Ning, "Knowledge-Based Program Analysis," *IEEE Software*, vol. 7, no. 1, pp. 74-81, Janurary, 1990.

[77] Mary Jean Harrold, Loren Larsen, John Lloyd, David Nedved, Melanie Page, Gregg Rothermel, Manvinder Singh, and Michael Smith, "Aristotle: A System for Development of Program Analysis Based Tools," in *Proceedings of the ACM 33rd Annual Southeast Conference*, 1995, pp. 110-119.

[78] Matthew Hetch, *Flow Analysis of Computer Programs*: Elsevier North-Holland, 1977.

[79] Susan Horwitz, Alan J. Demers, and Tim Teitelbaum, "An Efficient General Iterative Algorithm for Dataflow Analysis," *Acta Informatica*, vol. 24, no. 6, pp. 679-694, 1987.

[80] Susan Horwitz, Thomas W. Reps, and David Binkley, "Interprocedural Slicing Using Dependence Graphs," presented at PLDI, pp. 35-46.

[81] Joseph Hummel, Alexandru Nicolau, and Laurie J. Hendren, "A Language for Conveying the Aliasing Properties of Dynamic, Pointer-Based Data Structures," *IPPS*, pp. 1994, 1994.

[82] Mamdouh H. Ibrahim and Fred A. Cummins, "Objects with Logic," in *ACM Conference on Computer Science 1990*, 1990, pp. 122-133.

[83] Doug Janzen and Kris De Volder, "Navigating and querying code without getting lost," presented at AOSD 2003, Boston, MA USA, pp. 178-187.

[84] James Jenista and Brian Demsky, "Disjointness Analysis for Java-Like Languages," 2009.

[85] Neil D. Jones and Flemming Nielson, "Abstract Interpretation: a Semantics-Based Tool for Program Analysis," in *Handbook of logic in computer science: semantic modeling*, vol. 4, pp. 527-636, 1995.

[86] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser, "Generalized Symbolic Execution for Model Checking and Testing," presented at Tools and Algorithms for the Construction and Analysis of Systems, Warsaw, Poland, pp. 553-568, April 7-11, 2003.

[87] Greger Kiczales, Jim des Rivieres, and Daniel G. Bobrow, *The Art of the Metaobject Protocol*. Cambridge, Massachusetts: MIT Press, 1991.

[88] Greger Kiczales and Andreas Paepcke, *Open Implementations and Metaobject Protocols*: Xerox Corporation, 1994.

[89] Gregor Kiczales, John Lamping, Cristina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail C. Murphy, "Open Implementation Design Guidelines," in *International Conference on Software Engineering 1997*. Boston, Massachusetts: ACM Press, 1997.

[90] Michael Kifer, "Deductive and Object Data Languages: A Quest for Integration," presented at Proceedings of the International Conference on Deductive and Object-Oriented Databases, pp. 187-212.

[91] Michael Kifer, Georg Lausen, and James Wu, "Logical Foundations of Object-Oriented and Frame-Based Languages," *Journal of the Association for Computing Machinery*, vol. 42, no. 4, pp. 741-843, 1995.

[92] James C. King, "New Approach to Program Testing," *Programming Methodology*, pp. 278-290, 1974.

[93] James C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385-394, 1976.

[94] Paul Klint, Ralf Lämmel, and Chris Verhoef, "Toward an Engineering Discipline for Grammarware," *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 3, pp. 331-380, 2005.

[95] Ralf Kneuper, "Symbolic Execution: A Semantic Approach," *Science of Computer Programming*, vol. 16, pp. 207-249, 1991.

[96] Donald E. Knuth, "On the translation of languages from left to right," *Information and Control*, vol. 8, no. 6, pp. 607-639, December, 1965.

[97] Gordon B. Kotik and Lawrence Z. Markosian, "Automating software analysis and testing using a program transformation system," presented at Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification, pp. 74-84.

[98] Jens Krinke, Mirko Streckenback, Maximilian Storzer, and Christan Hammer, "Using Program Analysis Infrastructure for Software Maintenance," *Universität Passau*, March, 2003.

[99] David A. Ladd and J.Christopher Ramming, "A*: a Language for Implementing Language Processors," *IEEE Transactions on Software Engineering*, vol. 21, no. 11, pp. 894-901, 1995.

[100] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel:, "Context-sensitive program analysis as database queries," presented at PODS 2005, pp. 1-12.

[101] Chris Lattner and Vikram Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," presented at International Symposium on Code Generation and Optimization, San Jose, California, March 20-24.

[102] B. P. Leintz and E. Burton Swanson, *Software Maintenance Management*: Addison-Wesley, 1980.

[103] Thomas Lengauer and Robert Endre Tarjan, "A Fast Algorithm for Findinge Dominators in a Flograph," *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 1, pp. 121-141, July, 1979.

[104] Yuan Lin, Richard C. Holt, and Andrew J. Malton, "Completeness of a Fact Extractor," in *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*, 2003.

[105] Mark A. Linton, "Implementing Relational Views of Programs," in *Software Engineering Symposium on Practical Software Development Environments*, 1984, pp. 132 - 140.

[106] Panos E. Livadas and Scott D. Alden, "A Toolset for Program Understanding," in *Proceedings of the 2nd Workshop on Program Comprehension*. Capri, Italy, 1993, pp. 110-118.

[107] Rupak Majumdar and Koushik Sen, "Hybrid Concolic Testing," in *29th International Conference on Software Engineering*. Minneapolis, MN, USA: IEEE Computer Society, 2007, pp. 416-426.

[108] Michael Martin, Benjamin Livshits, and Monica S. Lam, "Finding Application Errors and Security Flaws Using PQL: a Program Query Language," presented at Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, San Diego, CA, USA.

[109] Anneliese von Mayrhauser and A. Marie Vans, "Program Understanding – A Survey," Colorado State University August 1994.

[110] Anneliese von Mayrhauser and A. Marie Vans, "Program Understanding Behavior During Adaptation of Large Scale Software," *IWPC*, 1998.

[111] John McCarthy, "Programs with Common Sense," presented at Proceedings of the Teddington Conference on the Mechanization of Thought Processes, pp. 75-91.

[112] minix-Documentation, "www.raspberryginger.com/jbailey/minix/html/tr_8c-source.html"

[113] Markus Mock, "Dynamic Analysis from the Bottom Up," in *Icse Workshop On WODA 2003 ICSE Workshop on Dynamic Analysis ICSE'03*, 2003, pp. 13-16.

[114] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers, "Dynamic Points-to Sets: a Comparison with Static Analyses and Potential Applications in Program Understanding and Optimization," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. Snowbird, Utah, United States, 2001, pp. 66-72.

[115] Markus Mohnen, "A Graph-Free Approach to Data-Flow Analysis," *CC*, pp. 46-61, 2002.

[116] Chris Moss, *Prolog++ The Power of Object-Oriented and Logic Programming*: Addison-Wesley, 1994.

[117] Gail C. Murphy and David Notkin, "Lightweight Lexical Source Model Extraction," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 3, pp. 262-292, July, 1996.

[118] Flemming Nielson, "Perspectives on Program Analysis," *ACM Comput. Surv.*, vol. 28, no. 4, 1996.

[119] Andreas Paepcke, *Object-Oriented Programming The CLOS Perspective.* Cambridge, Massachusetts: MIT Press, 1993.

[120] Santanu Paul, "SCRUPLE: a Reengineer's Tool for Source Code Search," in *CASCON*, 1992, pp. 329-346.

[121] Santanu Paul and Atul Prakash, "A Framework for Source Code Search Using Program Patterns," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 463-475, 1994.

[122] Santanu Paul, Atul Prakash, Erich Buss, and John Henshaw, “Theories and Techniques of Program Understanding,” presented at Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research, pp. 37-53.

[123] D. J. Robson, K. H. Bennett, B. J. Cornelius, and M. Munro, “Approaches to Program Comprehension,” *Journal Systems Software*, vol. 14, pp. 79-84, 1991.

[124] Spencer Rugaber, “Program Comprehension,” in *Encyclopedia of Computer Science and Technology Vol. 32, No.20*: Marcel Dekker, Inc., pp. 341-368, 1995.

[125] Koushik Sen, Darko Marinov, and Gul Agha, “CUTE: a concolic unit testing engine for C,” presented at Proceedings of the 10th European Software Engineering Conference, Lisbon, Portugal, pp. 263-272, September 5-9, 2005.

[126] Josep Silva and Olaf Chitil, “Combining Algorithmic Debugging and Program Slicing,” in *PPDP'06*. Venice, Italy: ACM, 2006, pp. 157-166.

[127] Janice Singer and Timothy C. Lethbridge, “What's so great about `grep'? Implications for program comprehension tools,” 1997.

[128] Guy L. Steele, *Common LISP: The Language*: Digital Press, 1990.

[129] Bjarne Steensgaard, “Points-to Analysis by Type Inference of Programs with Structures and Unions,” *CC*, pp. 136-150, 1996.

[130] Bjarne Steensgaard, “Points-to Analysis in Almost Linear Time,” *POPL*, pp. 32-41, 1996.

[131] Maximilan Stoerzer and Stefan Hanenberg, “A Classification of Pointcut Language Constructs,” presented at SPLAT'05: Workshop on Software-Engineering Properties of Languages and Aspect Technologie, Chicago, Illinois, March 14-18, 2005.

[132] Margaret-Anne D. Storey, “Theories, Tools and Research Methods in Program Comprehension: Past, Present, and Future,” *Software Qual J,* vol. 14, pp. 187-208, 2006.

[133] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller:, “How Do Program Understanding Tools Affect How Programmers Understand Programs? ,” in *WRCE '97*, 1997.

[134] Hisao Tamaki and Taisuke Sato, “OLD Resolution with Tabulation,” in *ICLP 1986*, 1986.

[135] Scott R. Tilley and Dennis B. Smith, “Coming Attractions in Program Understanding,” December 1996.

[136] Scott R. Tilley, Dennis B. Smith, and Santanu Paul, “Towards a Framework for Program Understanding,” in *WPC 1009*, 1996.

[137] Frank Tip, “A Survey of Program Slicing Techniques,” *Journal of Programming Languages*, vol. 3, no. 3, 1995.

[138] Peter Wegner, “Introduction and Overview,” in *Research Directions In Software Technology*, Peter Wegner, Ed. Cambridge, Massachusetts: MIT Press, pp. 1-36, 1980.

[139] Mark Weiser, “Program Slicing,” *ICSE*, pp. 439-449, 1981.

[140] Mark Weiser, “Programmers use slices when debugging,” *Communications of the ACM*, vol. 25, no. 7, pp. 446 - 452, 1982.

[141] Christopher A. Welty, *An Integrated Representation for Software Development and Discovery.* RPI Computer Science Dept., Rensselaer Polytechnic Institute, 1995.

[142] Reinhard Wilhelm, “Program Analysis - A Toolmaker's Perspective,” *ACM Comput. Surv.*, vol. 28, no. 4, 1996.

[143] Roel Wuyts, “Declarative Reasoning about the Structure of Object Oriented Systems,” presented at TOOLS USA 1998, Santa Barbara, California, pp. 112-124, August 3-7, 1998.

[144] Roel Wuyts, *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* Programming Technology Department, Vrije Universiteit, 2001.

[145] Ru-Gang Xu, *Symbolic Execution Algorithms for Test Generation.* Computer Science, University of California, Los Angeles, 2009.

[146] Michal Young and Richard N. Taylor, “Rethinking the Taxonomy of Fault Detection Techniques,” in *ICSE*, 1989, pp. 53-62.