MSc Individual Project

C Slicer
Project Report

Name: Samer Hamood
Year: 2004/5
Course: MSc Advanced Software Engineering
Department: Department of Computer Science
Student ID: 0324484/1
Supervisor: Mark Harman
Submission: 02/09/2005

King's College London
University of London

Contents Page

1. Technical Background

1.1 Program Slicing in the field of Software Engineering

Program slicing has multiple roles in the field of software engineering. As a technique, it has many useful applications at different stages of the software development life cycle as well as some that extend beyond this life cycle. In fact, almost every stage of the software engineering process can benefit from program slicing once code is available to slice. Its ability to assist in performing tedious and error prone tasks automatically, such as program debugging, testing, parallelization, program comprehension, program analysis, software maintenance, software metrics, software quality assurance, and program integration, gives it great importance and significance when engineering software [BG96, DHS95].

The field of program slicing is continually expanding, and new applications for it are constantly being found, however, program slicing like software engineering in general has by no means matured. Improved solutions to previous slicing problems continue to pop up ever so often containing new or updated algorithms and/or data structures, and pointing out the limitations and faults of the previous solutions. In this section, some of the applications of slicing previously mentioned will be discussed in overview.

1.1.1 Program Slicing: A debugger's point of view

The initial idea of program slicing came about when its inventor, Mark Weiser, noticed programmers debugging programs, and came to the conclusion that programmers naturally slice programs mentally when they attempt to debug them [Wei79a, Wei82, LW86]. Aiding this mental debugging became the motivation behind the development of an automated technique that would assist debugging activities by isolating the fault to a number of lines or statements in the program, with one or more of them actually containing the fault. This would mean that code that wasn't relevant to a set of variables the fault originated from, and therefore could not have caused the fault, would be omitted from the program, which would allow the debugger to concentrate on the lines or statements that are significant to the fault, and thus help in pinpointing it. Dynamic slicing in particular is appropriate when applying program slicing to debugging, as it produces smaller slices and makes available the inputs that caused the fault [HH01].

1.1.2 Testing

Slicing helps decompose programs, and for the purpose of testing, it makes test work faster and more efficient. By slicing particular slice criteria, inter-related modules can be identified, which then can be tested separately from the rest of the program. Because program slicing helps in understanding programs by dividing it into slices, the task of testing can be allocated to a number of testers, each with their own program domain to test [HD95].

1.1.3 Program Comprehension

Most, if not all, of the many useful applications of program slicing will require a program or at least a section of it that has some semantic significance, to be comprehendible. This is especially true in the comprehension phase of both software maintenance and re-engineering, particularly where legacy systems are concerned, since they do not necessarily need or have available documentation explaining the system or the original developers of it.

Conditioned slicing was used by De Lucia, Fasolino and Munro [DFM96] as a tool to aid program comprehension. Another similar technique called constrained slicing was introduced by Field, Ramalingham and Tip [FRT96]. Both utilized conditions to highlight cases of interest within a given program. Harman and Danicic [HD97] suggested that programmers can apply conditioned slicing combined with amorphous slicing to facilitate program comprehension and analysis. The conditions would be used to highlight some case of interest while the amorphous slicing would simplify the statements by removing the irrelevant parts, which would leave attention focused on the cases of interest [HH01].

1.1.4 Program Analysis

The process of program slicing is based on control flow and data flow analysis, and involves analysing a program's control and data flow to determine the correct decomposition of it with respect to the slice criteria [Wei84].

1.1.5 Software Maintenance

It is said that the maintenance and evolution phases of the software development lifecycle require some where around 70% of the total costs, so speeding up the process should lead to greater efficiency, which is exactly what slicing does when applied accordingly. Gallagher [Gal90] demonstrates that a slice can be constructed for each variable of the program irrespective of line position, covering all instances of the variable in the program, which was called a *decomposition slice*. These slices would in turn be used to partition the software with the intention to minimize the impact of software changes while performing maintenance on it. Because a decomposition slice would essentially tell us what code does and does not affect a variable, the programmer can alter software without causing unintentional changes due to the ripple effects of the software changes [HH01].

1.1.6 Software Metrics

With the discovery of different types of slicing, new applications were also being discovered. Among them was the ability to use slicing to help measure the cohesiveness of software. Cohesion, in brief, measures how strong elements of a module are related to each other. Functional cohesion refers to functions that should perform related tasks. Ott and Bieman [OB98] showed that static syntax-preserving slicing can help in measuring how cohesive a program is. A classification by Ott and Thuss [OT89] gives a class of cohesion by comparing slices with respect to different output variables.

When taking several slices from a function using a different variable for each, and analysing them by comparing the slices to each other on the bases of how much code they had in common, it is fair to say that these variables are connected in some manner. If we also decide that the function's tasks are captured by the computation performed on these variables, it is logical to deduce that the tasks were strongly related and thus concluding that the function was highly cohesive [HH01].

1.1.7 Software Quality Assurance

Software quality assurance auditors are faced with a myriad of difficulties, ranging from inadequate time to inadequate computer aided software engineering (CASE) tools. One particular problem is the location of safety critical code that may be interleaved throughout the entire system. Moreover, once this code is located, its effects throughout the system are difficult to ascertain. Program slicing is applied to mitigate these difficulties in two ways. First, program slicing can be used to locate all code that contributes to the value of variables that might be part of a safety critical component. Second, slicing based techniques can be used to validate functional diversity (i.e., that there are no interactions of one safety critical component with another safety critical component and that there are no interactions of non safety critical components with the safety critical components).

A design error in hardware or software, or an implementation error in software may result in a Common Mode Failure of redundant equipment. A common mode failure is a failure as a result of a common cause, such as the failure of a system caused by the incorrect computation of an algorithm. For example, suppose that X and Y are distinct critical outputs and that X measures a rate of increase while Y measures a rate of decrease. If the computation of both of the rates depends on a call to a common numerical differentiator, then a failure in the differentiator can cause a common mode failure of X and Y.

One technique to defending against common mode failures uses functional diversity. Functional diversity in design is a method of addressing the common mode failure problem in software that uses multiple algorithms on independent inputs. Functional diversity allows the same function to be executed along two or more independent paths.

1.1.8 Program Integration

The problem of integrating multiple versions of programs, known as program integration, is another interesting application of program slicing. Formalized by Horwitz, Prins, and Reps [HPR88], program integration is the problem of finding a program, let's say *D*, whereby given the programs *A* and *B* that are variants to a third program *C, D* is a program such that a) it "preserves the changed behaviour" of *A* and *B* with respect to *C* and b) it "preserves the preserved behaviour" of *C* in both *A* and *B*. In the case that such a program (program *D*) doesn't exist, *A* and *B* are said to *interfere* with *C*, which should be detected by the algorithm.

1.1.9 Re-engineering

Isolating desired functionality to be 'salvaged' and reused is commonly done as a re-engineering activity. Conditional slicing has been applied to reuse and was taken further by Cimitile, De Lucia and Munro [CDM96], as part of the RE-squared reverse engineering project. The comprehension phase of re-engineering can be aided by dissecting the code into slices for the sole purpose of understanding what the code's functions are, how it carries out these functions, and what elements are employed to perform them.

1.2 Program Slicing [HH01]

Program Slicing is a decomposition technique [Wei84], which identifies the parts of a program that (potentially) have semantic significance to a chosen point of interest, know as the *slice criterion*, extracting these parts to form the *program slice*, which is usually a reduced version of the original program. Slicing reduces programs to statements relevant for partial computation, deleting irrelevant statements.

The slice criterion is defined in terms of a set of variables, referred to as the *slice set*, and a position (a line number or statement) in the program. A program slice consists of all the program statements affecting the variable(s) at a particular position in the program, which are both specified by the slice criteria. For example, the slicing criteria S(a, 10) is a slice including all statements affecting the value of a at line 10.

The following section is a summary of some of the various slicing paradigms that exist today. As previously mentioned, the subject has not matured and is still very much in its evolutionary stage, nevertheless, there is a considerable amount of material on hand, some of which will be cited throughout the report.

1.2.1 Types of slicing

There are many different variants of slicing; among them are static slicing, dynamic slicing, forward slicing, backward slicing, condition or quasi-static slicing, syntax-preserving slicing, and amorphous slicing.

It is possible to split the slicing paradigms up into two main categories: semantic and syntactic. The semantic paradigms include the static, dynamic, and conditioned while the syntactic paradigms include syntax-preserving and amorphous slicing. The semantic and syntactic elements are two important aspects of slicing. The semantic element describes what part of the program is to be preserved. With the syntactic element, there are two possible types of slicing: 1) the program's original syntax can be preserved, removing sections of the program that have no affect on the semantics of interest, and 2) syntax transformations take place, which preserve the semantic detail of the program.

1.2.2 Static Slicing

The original slicing paradigm, static slicing is a simple and commonly used tool for constructing program slices. All other forms of slicing can be considered as an extension to static slicing.

A program slice [Wei84, HRB88] is produced by deleting the statements that have no semantic significance to the set of variables, know as the slice set, at the chosen point in the program. Thus, given a set a variables *V* at a specific point of interest *n*, a slice can be constructed for the variables in *V* at point *n*. After selecting a slicing criterion, we have a choice of two forms of slice: a backward slice or a forward slice. The difference between the two slices is that a backward slice contains the program statements that have an *effect* on the slicing criterion, whereas a forward slice contains the program statements that are *affected* by the slicing criterion.

Static slices tend to be rather large, especially when highly cohesive programs are sliced. This is because high cohesion occurs in programs where the computation performed on each variable is dependant on many other variables. However, slices constructed by static slicing do simplify the program, which helps in debugging activities.

1.2.3 Dynamic Slicing

Static slicing constructs slices for variables at a specified point in the program of interest. Some of those variables have a possible range of inputs that determine, in many cases, the behaviour or the output of a program. During the execution of a program, it may be that a value inputted causes some unexpected result, either in the form of incorrect values being outputted or program termination or both.

A dynamic slice [KR98] takes advantage of the available information about the input sequence supplied to a program at a specific execution. This dynamic information is extremely valuable when considering the presence of faults or bugs in a program. In the event of a failure occurring, the input to a program is available and can be used to locate the source of that failure. The slice will then only contain those statements that could have caused the failure within a specific execution of interest.

The static slice criterion together with the dynamic information (the sequence of input values for a variable), make up the ‘dynamic slicing criterion’, so given a variable *v* at a specific point of interest *n* on an input *i*, we can construct a dynamic slice. It is clear that dynamic slices are superior to static slices when the application is debugging, but this does not mean that we no longer need static slicing. Other applications, such as code re-use, require slices to be consistent for every possible execution. Since both slicing techniques have their advantages and disadvantages according to their applications, there is a trade-off between the static and dynamic paradigms, with static slices being larger but catering for all possible executions and dynamic slices being smaller but only catering for a single input.

1.2.4 Conditioned Slicing

So far, we have talked about two ways of constructing a slice, with one (static) providing no information about the input to the program, and the other (dynamic) providing specific information about the input. Conditioned slicing [CCD98, DF+00] bridges the gap between the two previous paradigms by using a boolean condition to specify a range of inputs rather than precise values.

The conditioned approach has been applied successfully to problems related to program understanding. Slicing with respect to conditions tells us a lot about how a program behaves under those conditions. Each conditioned slice contains the statements that capture various aspects of a program’s behaviour when the conditions stated are satisfied.

1.2.5 Amorphous Slicing

The static, dynamic, and conditioned approaches to slicing are all 'syntax-preserving'. By this we mean that the syntax of the slices is the same as that of the original program from which the slices were constructed. Therefore, these slices could be thought of as syntactic subsets of the original program. The only transformation that may have taken place when constructing the slices would have been the deletion of the statements that were irrelevant to the slice criterion.

Amorphous slicing [HD97, BH+00] transforms the program syntactically to simplify it, preserving the semantic behaviour with respect to the slicing criterion. An amorphous slice is never larger than its syntax-preserving equivalent and is often significantly smaller. Because amorphous slicing has the ability to extract the semantics of interest from a program by transforming the syntax, it can aid program comprehension, analysis, and re-use.

1.2.6 Scope of slicing

Program slicing can be intra-procedural or inter-procedural [HPR88, HRB88], and in recent years has also been applied to object-oriented programs for object, class, and even interface slicing [Tip96, LH96, Ste98, BE93]. Other applications include concurrent programs [Jin93, Jia99, ZCU96], logic programs [SD96, KNN99, ZCU97], functional programs [JH99], and specifications [WA98].

1.2.7 Variations on slicing

Chopping and dicing are two program isolating techniques that are variations on the slicing theme.

Chopping [JR94, CC93] identifies dependencies between a *source* variable instance and a *sink* variable instance on the bases of a define-use relation. All variables are treated as part of this relation and are represented as nodes on a graph. The output of a chop is the statements that cause the definition of the *source* to affect the uses of the *sink*.

Dicing [LW86] is a fault localization technique introduced to further reduce the number of statements that need to be examined to find faults. Whereas a slice makes use only of information on incorrect variables at failure points, a dice also makes use of information on correct variables, by subtracting the slices on correct variables away from the slice on the incorrect variable. The result is smaller than the slice on the incorrect variable; however, a dice may not always contain the fault that led to a failure.

1.3 Slicing Algorithms

The development of programs that slice another program, or "slicers", is that of the development, and indeed, the evolution of slicing algorithms. Undesirably, these algorithms, in spite of the continual advancements and improvements made in the field, remain complicated due to the complex nature of programming languages, which require complex structures to represent them in a way applicable to slicing before some sort of slicing computation on these structures can be performed.

1.3.1 Brief History of Slicing Algorithms

In 1979, Mark Weiser published a thesis [Wei79b] that would change software engineering for the better by introducing the concept of slicing, presenting the first ever slicing algorithm, the data-flow equations algorithm, that deals with static intra-procedural program slicing and forms the basis for many slicing algorithms today. In later years, other slicing algorithms were developed using alternative approaches and instruments. In the 1984 Ottenstein paper [OO84], the authors acquainted us with the notion of slicing using a Program Dependency Graph (PDG) that transformed the way we looked at program procedures and their intra-procedural dependencies forever.

1988 saw the publication of the first paper on the System Dependency Graph (SDG) [HRB88] by Horwitz, Reps and Binkley, which took slicing to the next level by crossing the boundaries of procedure calls, expanding the slicing domain to the inter-procedural, and extending the range of sliceable program elements to include procedure specific information such as procedure arguments. After nine years, it was finally possible to slice an entire procedural program using Horwitz, Reps and Binkley's algorithm because it solved the inter-procedure slicing problem, handling procedures as well as intra-procedural data. The SDG was a huge break-through for the slicing community, and in 1996, was applied to Object-Oriented slicing in a paper written by Larsen and Harrold [LH96].

1.3.2 Slicing Algorithm Approaches

There exist several algorithms that solve the various slicing problems in reasonable time, but for the purpose of this report only static slicing algorithms will be considered, although, the algorithms for the other slicing paradigms use the same principles and are merely augmentations. In general, slicing is achieved through one of three algorithmic approaches: 1) data-flow equations 2) system dependency graph 3) parallel algorithm; all based on the concept of control and data dependencies and are defined in terms of a graph representation of a program.

1.3.2.1 Graph Representations

For procedures without procedure and function calls, there are two choices of graph representation: 1) the control flow graph, or 2) the program dependency graph. An inter-procedural representation that takes into account procedure and function calls also exist with its own slicing algorithm and will be review in section 1.3.2.3.

Control Flow Graph

Control and data dependencies are represented in the form of the control flow graph (CFG) of a program [Hec77]. The CFG is made up of nodes that correspond to statements and control predicates, and edges between the nodes that define possible control flow. Two additional nodes (entry and exit) of the CFG refer to the start and the end of a program. Nodes in the CFG define and/or reference a set of variables. Various types of data dependencies can be established using such graphs, among them are flow dependence, output dependence, and anti-dependence [FOW87]. We will only talk about flow dependence as the others are irrelevant to slicing.

Flow dependence

If a value computed at statement *i* is used at statement *j* then it is said that *j* is *flow dependant* on *i* for a particular program execution. In other words, there exists a variable *x* that is defined by *i* and used by *j*, and there is a path from *i* to *j.* Another way of stating this is that the definition of variable *x* at node *i* is a *reaching definition* for node *j*.

Control Dependence

Control dependence describes the relation between a control predicate and the statements within branches of that control predicate, and is usually defined in terms of post-dominance. If all paths from a node *i* to the exit or stop node pass though a node *j* then it is said that *i* is *post-dominated* by *j*. So, if there is a path from *i* to *j* such that *i* is not post-dominated by *j* and *j* post-dominates all nodes in path *i* to *j* not including *i* and *j* then *j* is control dependent on *i.*

Control predicates 'control' [FOW87] the execution of the nodes within its body, determining whether or not control is to pass to these nodes. Given a predicate *p*, the set of nodes that depend on the outcome of *p* in this way are called the *controlled nodes* of *p*. For block structured languages, controlled nodes can easily be computed – the controller nodes form the body of the consequent and alternative branches of conditional statement predicate nodes or the body of loop predicate nodes. Ferrante, Ottenstein and Warren produced an algorithm for computing the controlled nodes for unstructured languages [FOW87].

Program Dependency Graph

A program dependency graph (PDG) is another graph representation of a program [OO84, FOW87, KK+81]. Like the CFG, the vertices correspond to statements and control predicates; however, in a PDG, the edges correspond to data and control dependences between the nodes. As the dependence edges have a partial ordering to them, the statements have to be executed in this order to maintain the program's semantics [HPR88a, HPR88b, HRB88].

1.3.2.2 Data-flow equations [DHS95, Tip95]

The original approach to program slicing defined by Weiser [Wei84], computes slices by solving iteratively a set of data-flow equations derived directly from the CFG of a program. An iterative algorithm is presented by Weiser using two levels of iteration that trace the following:

1. Transitive data dependences in the presence of loops in the program.
2. Control dependences, initiating the inclusion of control predicates for which each, step 1 is repeated to include the statements it is dependent upon.

Successive sets of *relevant variables* are computed for each node in the CFG, and from these, sets of *relevant statements* can be derived with the fixpoint of the latter set defining the slice. The first stage is determining the *directly relevant variables*, which is an instance of step one of the iteration process summarised above.

Defined and Referenced Variables

Node *i* of a CFG represents a program statement; a statement that may define and\or reference variables. The set of defined and referenced variables for each node *i* is denoted DEF(*i*) and REF(*i*) respectively. For example, take the assignment statement a = b + c; that node *i* represents, then DEF(*i*) = {a} and REF(*i*) = {b, c}.

Directly Relevant Variables

For a slice criterion (*V, n*), in the first iteration, the set of directly relevant variables, denoted $R_C^0(n)$, for the slice node *n* is the slice set *V*, and for all the other nodes is an empty set ($R_C^0(n) = V$, $R_C^0(m) = \varnothing, m \neq n$). The set of directly relevant variables for each node *i* is defined in relation to the set of directly relevant variables of all nodes *j* that have a direct edge to *i*, $i \rightarrow_{CFG} j$, in the CFG. $R_C^0(i)$ is made up of all the variables *v* such that either $v \in R_C^0(i)$ and $v \notin \mathrm{DEF}(i)$ or $v \in \mathrm{REF}(i)$ and $\mathrm{DEF}(i) \cap R_C^0(i) \neq \varnothing$.

The full equation for each edge $i \rightarrow_{CFG} j$ in the CFG is as follows:

$$R_C^0(i) = R_C^0(i) \cup \{v \mid v \in R_C^0(i), v \notin \mathrm{DEF}(i)\} \cup \{v \mid v \in \mathrm{REF}(i), \mathrm{DEF}(i) \cap R_C^0 \neq \varnothing\}$$

Directly Relevant Statements

From $R_C^0(i)$, deriving the set of relevant statements, denoted S_C^0, is possible. S_C^0 is the set of all nodes *i* that define a variable *v* that is relevant at the successor node of *i*.

$$S_C^0 = \{i \mid \exists j, i \rightarrow_{CFG} j \wedge \mathrm{DEF}(i) \cap R_C^0(j) = \varnothing\}$$

Indirectly Relevant Variables

The subsequent iterations of Weiser's algorithm involve considering control dependences. The referenced variables in the control predicate are *indirectly relevant* when at least one of the statements in its body is relevant. A branch statement *b* has what is known as a *range of*

influence INFL(*b*), which is the set of statements control dependant on it, or in other words, the controlled nodes.

Below is the equation that defines the branch statements B_C^k that are indirectly relevant because of the influence they have over nodes *i* in S_C^k. Afterwards, we have the equation that determines the *indirectly relevant variables* R_C^{k+1}.

$$B_C^k = \left\{ b \mid \exists i \in S_C^K, INFL(b) \right\}$$

$$R_C^{k+1}(i) = R_C^K(i) \cup \bigcup_{b \in B_C^K} R^0_{(b,\mathrm{REF}(b))}(i)$$

1.3.2.3 System Dependency Graph [HRB88, Tip95]

An alternative approach to slicing, presented by Horwitz, Reps, and Binkley [HRB88], defines an algorithm to compute accurate inter-procedural slices and comprises of the following three components:

1. System Dependency Graph (SDG), a graph representation of multi-procedural programs.
2. The computation of inter-procedural summary information, which consists of precise dependence relations between the input and output parameters of each procedure call, and is explicitly present in the SDG in the form of *summary edges*.
3. A two-pass algorithm that extracts inter-procedural slices from an SDG.

Parameter Passing

Using the Horwitz-Reps-Binkley algorithm [HRB88], parameter passing by value-result is performed as follows:

(i) the calling procedure copies its actual parameters to *temporary* variables before the call,
(ii) the formal parameters of the called procedure are initialized using the corresponding temporary variables,
(iii) before returning, the called procedure copies the final values of the formal parameters to the temporary variables, and
(iv) after returning, the calling procedure updates the actual parameters by copying the values of the corresponding temporary variables.

The algorithm also deals with parameter passing by reference as long as the problem of aliases is solved. There are two methods for dealing with aliases:

1. Transforming the original program into an equivalent alias-free program, or
2. Using generalized flow dependence that takes aliasing patterns into account [Bin93a].

Both methods resolve aliasing, however, the first produces more precise slices, whereas the second is more efficient. More information on parameter passing mechanisms is available in Aho, Sethi, and Ullman's paper [ASU86].

1. System Dependency Graph

SDGs consist of a PDG for the main procedure and a PDG for every other procedure. They also contain different types of vertices that PDGs do not. Call statements have a *call-site vertex* in the SDG in addition to *actual-in* and *actual-out* vertices that are control dependent on the call-site vertex and represent the copying of actual parameters to and from temporary variables. For each PDG, there is an entry vertex, and *formal-in* and *formal-out* vertices that are control dependent on the entry vertex and represent the copying of formal parameters to

and from temporary variables. Inter-procedural data flow analysis can be used [Ban79] to determine the set of variables that can be referenced or modified by a procedure. With this information, actual-out and formal-out vertices can be removed for parameters that will never be modified, which yields more precise slices.

Apart from the intra-procedural dependence edges already mentioned, the following inter-procedural dependence edges are also part of an SDG:

1. a control dependence edge between a call-site vertex and the entry vertex of the corresponding procedure dependence graph,
2. a *parameter-in* edge between corresponding actual-in and formal-in vertices,
3. a *parameter-out* edge between corresponding formal-out and actual-out vertices, and
4. *summary edges* that represent *transitive inter*-procedural data dependences.

2. Computation of Summary Information

The second part of the Horwitz-Reps-Binkley algorithm entails the computation of the summary edges between vertices for input\output parameters of a procedure. If the SDG contains summary edges then incoming values for the input parameter may be used to obtain the outgoing values for output parameters. The algorithm identifies summary edges by constructing an attribute grammar that works like a call graph, modelling the calling relationships between the procedures in the program. After the construction of the attribute grammar, the subordinate characteristic graph for this grammar is computed. This graph contains edges corresponding to precise transitive flow dependences between input/output parameters of each procedure in the program. The subordinate characteristic graph's summary edges are copied to the correct places at each call site in the SDG. Other algorithms for computing summary edges have been defined [Mic94, RHS94, RSH94] that are more efficient than Horwitz-Reps-Binkley algorithm.

3. Two-Pass Algorithm for Extracting Inter-procedural Slices

For the third part of the Horwitz-Reps-Binkley algorithm, a two-pass traversal of the SDG is carried out. The SDG's summary edges determined from the previous phase, act as a means of finding a way around the calling context problem. Suppose that slicing begins at a vertex *v*, the first stage determines every vertex from which *v* can be reached *without dropping into procedure calls*. Calls can be circumvented without dropping into them due to the transitive inter-procedural dependence edges of the SDG. In the second stage, the remaining vertices in the slice are determined by dropping into all previously circumvented calls.

The Horwitz-Reps-Binkley algorithm [HRB88] does not always compute slices that are executable programs. Situations where only a subset of the vertices for actual and formal parameters are in the slice, correspond to procedures whereby only some of the arguments are removed from the slice; depending on the call of the procedure, arguments may or may not be included. Horwitz, Reps, and Binkley propose two solutions to this problem that both transform a non-executable slice into an executable program. With the first approach [HRB88], a number of variations of a procedure may be integrated into the slice, which results in a slice that is no longer a restriction of the original program. As for the second [Bin93b], the slice is extended from all parameters present at some call to all calls that occur in the slice. The vertices that the added vertices are dependent on must also be added to the slice. The first approach yields smaller slices than the second.

1.3.2.4 Parallel Algorithm [DHS95]

The last approach to static slicing is intra-procedural and is the simplest of the three. Danicic, Harman, and Sivagurunathan [DHS95] present a parallel algorithm that defines a slice in terms of a set of recursion equations derived from the CFG of a program. Essentially, the algorithm is a parallel version of Weiser's (section 1.3.2.2), exploiting the natural parallelism in the CFG. The algorithm consists of converting the CFG into a process network analogous to that defined by Abramsky [Abr84]. The processes that form the network concurrently communicate to each other through message sending across the arcs\edges of the CFG.

Each message contains a set of variables, which are strongly related to the 'relevant variables' of Weiser's algorithm (section 1.3.2.2).

In addition to computing a slice, the algorithm also has several interesting properties that can be used for simultaneous slicing as well as generating equivalent slicing criteria. The algorithm consists of two stages once the CFG of the program being sliced is available:

1. The CFG is converted into a network of concurrent processes.
2. The network of concurrent processes is executed, the result of which is the set that contains the nodes corresponding to the relevant statements with respect to the slice criterion.

Process Networks

A network of concurrent processes can be modelled, as shown in [Abr84], as a graph with its nodes\vertices representing processes and its arcs\edges representing communication channels. The processes are defined as recursive functions on streams of data. These functions form sets of recursion equations, which define the behaviour of the network, and whose outputs represent the possibly infinite streams of data communicated between the processes.

The first phase of the algorithm consists of converting the program to be sliced into a CFG and then compiling it into a network of concurrent processes. The process network's topology is derived from the inverse of the program's CFG, known as the Reverse Control Flow Graph (RCFG) [ASU86], with the nodes of the CFG corresponding to the processes and the arcs of the RCFG corresponding to the communication channels. The incoming arcs of a node *i* correspond to inputs to a process *i*, and the outgoing arcs of node *i* correspond to outputs of process *i*. An outputted message is output on all output channels.

Process Behaviour

As indicated previously, messages are sets of variable names and node identifiers that are sent and received by processes. Each process's behaviour is dependent on the following information, which is obtained directly from the CFG of the program:

(i) *i*: The identifier of the node of the CFG.
(ii) REF(*i*): The set of variables referenced by *i.*
(iii) DEF(*i*): The set of variables defined by *i*.
(iv) C(*i*): The set of nodes controlled by *i*.

In the RCFG, predicate nodes (section 1.3.2.2) have more than one input and because the algorithm is defined for side-effect free languages, these nodes will not define any variables, i.e. DEF(*i*) = $\varnothing$. All other nodes, by definition, will not be predicates and will consequently not control any other nodes, i.e. C(*i*) = $\varnothing$. The specification for the behaviour of process *i* can be defined formally in CSP style notation [Hoa85] as follows:

$$
\begin{aligned}
P(i) = ?S \rightarrow (&\mathit{if}\ \ S \cap (\mathrm{DEF}(i) \cup C(i)) \neq \varnothing \\
&\mathit{then}\ !((S \setminus \mathrm{DEF}(i)) \cup \mathrm{REF}(i) \cup \{i\}) \\
&\mathit{else}\ !S\,); \\
&P(i)
\end{aligned}
$$

This definition states that if the input, denoted *S*, to process *i* has any elements that are also in either the set of defined variables of *i* or the set of controlled variables of *i* then process *i* outputs the set containing the following:

1. every input variable (elements of *S*) that it does not define,
2. every variable that it references,
3. its node identifier, *i*.

However, in the case where *S* has no elements that are also in either of the two sets mentioned (defined and controlled variables of *i*) then process *i* simply outputs *S*. The behaviour of process *i* is repeated for every input message it receives, with process *i* waiting for its next input.

From the definition above, we can conclude that each process *i* individually outputs a message according to its specific input. Therefore, process *i* can be thought of as a function from all its inputs to all its outputs. This idea of a process being a function is very important in the light of recursive functions that output a solution to the static slicing program.

Constructing the slice

The second phase of the algorithm deals with the construction of the slice. To construct the slice for a slice criterion (*V*, *n*), the network of concurrent processes is executed with network communication being initiated by outputting the message *V* from process *n*. It should be noted that if a process *i* outputs its node identifier, *i*, then the node *i* is to be included in the final slice. Network communication is terminated if any process of the network outputs the same message more than once. Consequently, the algorithm comes to an end, and nodes that correspond to statements that are to be kept in the final slice are obtained from the resulting output. The slice of the program is made up of the nodes whose identifiers are input to the entry node of the CFG, because the entry node is reachable via every node in the RCFG, which means messages outputted by all nodes eventually reach the entry process.

Functional Networks

The network of processes described above can be defined in terms of recursion equations over a finite set of variables and node identifiers. The behaviour of each process *i* is defined as a function *i* on sets as described for the processes:

$$F_i(S) = \textit{if } S \cap (\mathrm{DEF}(i) \cup C(i)) \neq \varnothing$$
$$\textit{then } (S \setminus \mathrm{DEF}(i) \cup \mathrm{REF}(i) \cup \{i\})$$
$$\textit{else } S$$

Functions, denoted F_i, take arguments, execute the computation defined above, and then return a result. Input to process *i* is represented by the argument to function *i*, and output to process *i* by the result of function *i*. Configuring the functions in different ways allow different network topologies to be put together. If a process has more than one input then the corresponding function's argument is the union of the individual inputs. For each cycle (loop) in the RCFG, there is a functional equation.

2 Project Objectives

The main objective of the project is the development of a simple program slicer that can construct backward static intra-procedural slices of a specific subset of the C language. Static backward slicing is a very common and easy way to slice, which walks backwards through the code, singling out the statements that are relevant to a given slice criterion (section 1.2). This slicing method will transform the inputted program from its original state to its static backward sliced state, which may or may not be different, depending on the program and the slicing criterion. Our main objective, outlined above, can be split up into following sub-objectives:

1. Constructing the slice.
2. Developing a user interface, as a framework to construct and display slices in.

But before we can construct a slice, we must first define the subset of the language (The C language in this project) that is going to be sliced.

2.1 C Language Subset

Obviously, it is not practical to slice the entire C language given the time and knowledge available, therefore we must restrict the range of what can be sliced to a subset of C. The selected subset of C contains the basic variables and their declarations, expression statements, selection statements (except `switch` statements), and iteration statements (except `do` statements). What will not be part of the subset are derived variables (arrays, functions, pointers, structures, and unions) and enumerations, the comma operator, the `sizeof` operator, the unary operators, labelled statements, jump statements, external definitions, macro definitions, the pre-processor features (file inclusions, conditional compilation, line control, error generation, pragmas, and null directives), procedures, and function and procedure calls.

The subset of C to be sliced is fairly general in that it includes the fundamental constructs/features found in Turing complete programming languages, and leaves out many of the language specific features. Syntactically, the code to be sliced should more or less look exactly the same in any C syntax languages (C++, Java, C#), and with the availability of a parser for one of those languages, slicing it merely becomes a matter of writing code to process the outputted AST to accommodate the desired subset.

2.2 Slice Construction

Due to its simplicity, and appropriateness with respect to the preferred slicing paradigm, the parallel algorithm approach to backward static slicing, described in section 1.3.2.4, will be the basis for solving the static slicing problem. In view of the chosen approach, constructing a slice can be thought of as a four stage process:

1. Converting code into an abstract syntax tree (AST).
2. Converting the AST into a RCFG.
3. Implementing a slicing algorithm using the RCFG.
4. Converting the slicing algorithm output back into the original program's syntax.

Both the first and last stages are fundamental to the slicing process and are constant no matter what slicing approach is taken.

2.2.1 Program to AST Conversion

The first stage is achieved with the aid of a C language parser that converts the inputted C program into its AST. A parser is a computer program that carries out the task of parsing, which is the process of analyzing a continuous stream of input (read from a file or a keyboard, for example) in order to determine its grammatical structure with respect to a given formal grammar. An AST is a tree data structure that is, for the purpose of slicing, used as a

compiler's or interpreter's internal representation of a computer program and is described by the abstract syntax of that program.

As a fully functioning C parser has already been provided, the stage is thus complete and will not be discussed further.

2.2.2 AST to RCFG Conversion

The second stage involves converting each program statement into a CFG node that can be applied to the slicing algorithm, and the relationships between the statements into CFG arcs. Compiling the AST into the RCFG of a program consists of two phases:

(i) Extracting the relevant code elements from the AST and converting them into an array of CFG nodes.
(ii) Representing the arcs of the RCFG.

Phase one requires that the AST of the program be traversed, extracting the code elements relevant to slicing that constitute a valid statement, and capturing control dependence information (section 1.3.2.1). For simple static slicing on C, these elements include variable and constant identifiers, statements, and some operators. In general, code elements relevant to slicing do not include storage classes, type specifiers, type qualifiers, comments, constants, or string literals.

Since the C parser converts the C code into an AST in Extendible Mark-up Language (XML) form, an XML parser will be employed to traverse the AST. The Document Object Model (DOM) is an XML parser that looks at an XML document as a series of parent and child nodes of different types that contain their attributes. These nodes are captured as the XML document is parsed, allowing XML code to be traversed as a tree structure. DOM provides an API that can access each individual node and its attributes, in addition to performing other operations on XML documents. To parse an XML document, the name of the file that corresponds to the document is passed as an argument to the parser object.

Phase two depends on the successful completion of stage one so as to be able to compute an accurate graph representation of the program. Modelling the arcs of the CFG is relatively easier for a simple configuration of statements but gets complicated when a program contains multiple compound statements because of the branching of the control paths. Finally, the matter of reversing the CFG is to be addressed, the solution of which is well known in graph theory.

2.2.3 The Algorithm

One of the sub-objectives of the project is to implementation the parallel algorithm (section 1.3.2.4) for static intra-procedural program slicing, which has been chosen for the purpose of computing slices for two reasons: 1) it is the simplest approach out of the three choices and, 2) it computes basic backward static intra-procedural slices.

2.2.4 Original Syntax Conversion

The last stage of the slicing process is converting the output of the algorithm back into the syntax of the original program. Fortunately, the problem has been resolved and is reviewed in [HRB88, HR92, Wei79b, Wei84].

2.3 User Interface

A C file is to be entered at the command line and evaluated for its validity; if the code has no errors, it will be passed to the slicer mechanism. The slicer mechanism is responsible for transforming the original code into the sliced version by constructing a slice in the way described in section 2.2. Before slicing can commence, a slice criterion needs to be specified. The user specifies a slice criterion by variable(s) and statement through inputting those values, which will in turn be inputted to the slicing mechanism. The code will then be

redisplayed after the slicing process is complete, which should result in the sliced version of the code being outputted on the screen.

2.4 Desired Outcome

On providing the proposed slicer program the program below and on entering the slice criterion ({count}, 11),

Lines	Program
	`void main()`
	`{`
(1)	`pass = 0;`
(2)	`fail = 0;`
(3)	`count = 0;`
(5)	`while (count <= 90)`
	`{`
(7)	`if (Marks >= 40)`
(8)	`pass++;`
(9)	`if (Marks < 40)`
(10)	`fail++;`
(11)	`count++;`
	`}`
	`return 1;`
	`}`

Figure 1: Original Program

the slice of the program above should be produced and outputted, looking something like the following:

Lines	Program
	`void main()`
	`{`
(1)	
(2)	
(3)	`count = 0;`
(5)	`while (count <= 90)`
	`{`
(7)	
(8)	
(9)	
(10)	
(11)	`count++;`
	`}`
	`return 1;`
	`}`

Figure 2: Sliced Program

3. Technical Contribution

3.1 Analysis

When examining the resources available, especially the Danicic-Harman-Sivagurunathan paper [DHS95], which is the main source of information on intra-procedural backward slicing and is the basis for slicing in this project, a number of prominent components were identified with some being essential to the process of slicing. As a result, functional and data requirements have also been identified, allowing for a requirements specification to be defined.

3.1.1 Data Requirements

The data requirements can be defined in terms of the different inputs and outputs involved in the slicing process. On analysing the relevant slicing material, the following inputs and outputs have been identified:

3.1.1.1 Inputs

Slicer (entered by the slicer user)

- C program file
- Slice criterion - set of variables (usually one) and a number

Software (entered by the software components)

- AST
- RCFG
- Node identifiers – number identifying a node.
- Referenced variables – set of variables referenced by a program statement
- Defined variables – set of variables defined by a program statement
- Controlled node identifiers – set of node identifiers controlled by a statement.

3.1.1.2 Outputs

Slicer

- Sliced version of C program file
- Variables of the program
- Slice criterion

Software

- RCFG

Some of the data identified are composites and require decomposing to be better understood, and others are atomic and may form part of a composite. Composite modules can be described as data structures that may also perform functions on the data they contain.

3.1.2 Functional Requirements

In terms of visible functionality, the slicer will take as input a C program file and slice criterion, and from these construct a slice, producing a new program as output. The slicer should also output the slice criterion and a list of the inputted program's variables. Section 2.2 briefly explains how a program is sliced; our next step is to break down the stages into more comprehendible and manageable tasks.

3.1.2.1 AST to RCFG Conversion

Given that a CFG is composed from CFG nodes which are then compiled into network processes, and that these nodes are configured in a particular arrangement to represent a program, we can break up the AST to RCFG conversion stage into two tasks:

1) Converting the AST into CFG nodes and,
2) Configuring the nodes into a CFG program representation.

Conversion of AST into CFG Nodes

The first task is essentially finding the right data and placing them in the right place holder (data structure) to be made use of at a later stage, namely the second task, and thus the first task must be complete before we can begin with the second. To simplifying this process further, let us divide the task up into the following sub-tasks:

- Traverse the AST.
- Extract the values from the AST (accessing values stored in XML tags using DOM).
- Pass extracted values to statement components.
- Pass the statement component to a CFG node.
- Pass controlled CFG nodes to their controller CFG nodes.

CFG Nodes Configuration into CFG

By analysing the structure of a CFG, it becomes clear that its node configuration is dependant on two factors; i) the position of the CFG node and, ii) the last controlled CFG node of each branch of a predicate CFG node. Both factors, identified from the preceding task, are vital to the CFG nodes' transformation from ordinary nodes that have no direct relationship with each other to nodes that are linked together by the correct arcs. A correct arc is, as will be seen subsequently, determined by how a CFG represents a program.

Going by the slice subset specified in section 2.1, the CFG that we intend to construct has three control structures:

1. Sequence - a series of successive nodes from 1 to n

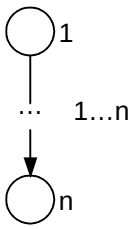

Figure 3: CFG representation of a sequence of statements

2. Selection\Conditional Statements – a node with a consecutive and alternative branch.

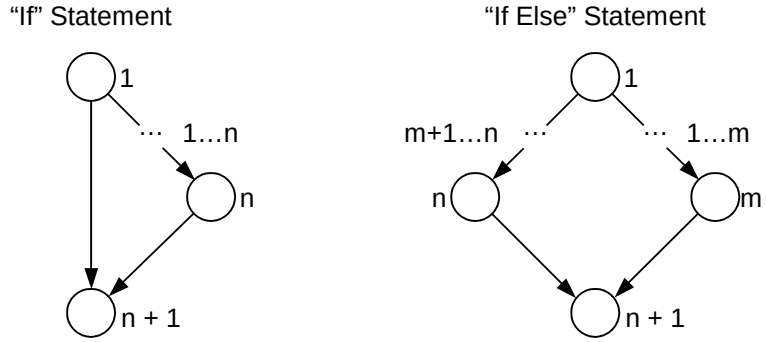

Figure 4: CFG representation of selection statements

3. Iteration statement – a node with one branch, controlling all nodes within it.

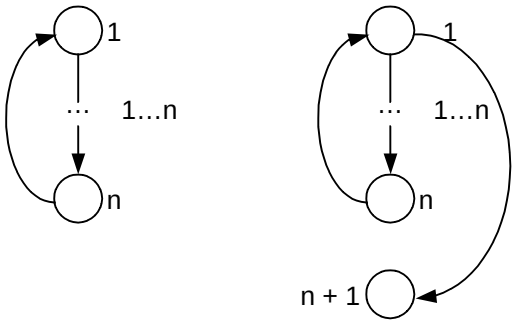

Figure 5: CFG representation of iteration statements

Reversing the CFG has the affect of changing the direction of the arrows in the diagrams so that they point the opposite way. Modelling these three structures and the possible combinations of them will be the second task ahead of us. This task may also be broken down into the following:

- Represent the individual and combined program properties (the three structure above) as links between nodes
- Reverse the CFG

1. Slicing Algorithm Implementation

The sequential implementation of the slicing algorithm is, in fact, a variation on the original algorithm that instead of defining the slice computation in terms of recursion equations, defines it as a graph data structure iteration using process networks derived from the RCFG rather than functional networks.

The algorithm traverses the RCFG, inputting the outputs (behaviour computation) of processes into every other process it has an outgoing arc to, computing the union of the individual inputs to processes with more than one input, and iterating in the presence of cycles (loops). Each process *i* of the network is represented by CFG node *i* that contains the set of variables defined, DEF(*i*), along with the set of variables referenced, REF(*i*), by the corresponding statement *i* in the program to be sliced. These values are extracted from the AST as part of the CFG node compilation stage that precedes the execution of the slicing

algorithm. The behaviour of the each process is that defined in section 1.3.2.4. The algorithm runs through the RCFG, processing each node until there are no new labelling on the edges (i.e. no new outputs). The algorithm's result is the output of the first node in the CFG.

2. Converting slice result into original program's syntax

Converting the code captured by a program statement entity back into the syntax of the original program is achieved by viewing the program as a source map [HRB88, HR92, Wei79b, Wei84]. For each program statement, the syntax is divided into its individual parts, or tokens - a word or an atomic element within a string, and is then searched for in the original file. The exact position of each token must be known to retrieve the program statement as it appears in the original program. Once the positions of the statement tokens are found within the original program's file, they are stored, linked in some way to the CFG node of the statement, and later used to obtain the syntax of the original program. To summarise, below are the sub-tasks of this stage:

- Read original program file locating token positions
- Store positions for all statement tokens
- Write to a new program file

It should be noted that seeing as an AST doesn't cater for braces and parentheses (as well as other such syntax rules), it means that these syntactic elements are a special case and have to be accommodated if they exist in the original program to preserve its syntactic appearance.

3.1.3 Derived Data Structures

From the list of inputs and outputs (section 3.1.1), it is clear that the CFG is an adjacency graph data structure that holds data representing nodes (vertices) and arcs (edges), and may execute operations on that data such as computing in\out-degrees of the edges, or iterating through them. To a lesser extent, the slice criterion is also a data structure composed of a set of variables and a number that represents a position in the program.

In a CFG, each node corresponds to a program statement and is identified by a number, the node identifier, which coincides with the statement's sequential order with respect to the other statements in the program. Node identifiers are atomic values that have no meaning as independent entities. In fact, they can be considered data members of a CFG node data structure.

The referenced and defined variables as well as the controlled node identifiers are not composite structures but collections of atomic values, which have in common the fact that they are all input values for the slicing algorithm. In the Danicic-Harman-Sivagurunathan algorithm (section 1.3.2.4), a *network process* has been defined as an element made up of these three collections, using them to compute a value (the process output). For that reason, a network process can also be considered a data structure with its behaviour being one of its functions.

According to the algorithm, CFG nodes are to be compiled into network processes, telling us that there is some relationship between the two entities. Seeing as the CFG nodes and network processes are similar - a network process basically replaces a CFG when applying the slicing algorithm on the RCFG, the CFG node can be thought as a network process.

So far we have identified four data structures, two of which have been obtained from some of the input data. However, in addition to these data structures, others may also be discovered by further analysing the data requirements. For example, there exist different types of CFG nodes depending on the statements they correspond to. Computation nodes correspond to declaration and expression statements that define and reference variables, where as predicate nodes correspond to selection and iteration statements that reference variables and control other statements. These distinctions between CFG nodes imply a possible generalization of the CFG node data structure, i.e. the classification of cases, computation and predicate nodes.

Apart from the entities derived directly from the data requirements, supplementary data structures can be developed to capture program statements and their values from the AST generated by the C parser. Program statements hold data pertaining to expression statements, declarations statements, iterative statements (while and for loops), selection\conditional statements (if and if else), etc. The range of different statements having multiple structures each to model a specific statement type.

A program statement module intermediates between the AST and the CFG nodes, containing the data extracted from the AST that will then make up the key elements of a CFG node, i.e. the defined and referenced variables of each independent program statement. There is also a secondary function for these data structures. Since they hold the standard syntax necessary for each statement to run, which sadly isn't entirely available from the AST, as soon as the relevant statement data is obtained from the AST, a valid statement with the required valid syntax can be stored and used to obtain the program's original syntax from the source code file.

When converting the slice result back into the program's original syntax, a data structure can be utilized to store the position of a token; in turn, a collection of these would provide the positions of all the relevant statements in program.

At present, the following are that data structures that have been deemed as significant components in regards to the software development of the slicer:

1) Slice Criterion
2) Control Flow Graph
3) Network Process
4) Control Flow Graph Node
5) Computation Node
6) Predicate Node
7) Syntax
8) Token Point

Throughout the project life cycle we can expect to discover more data structures that tie into the software in some manner, or we may decide to add extra data structures to deal with specific functions. To see the class hierarchies, go to appendix 1.a.

3.2 Design

3.2.1 Software Architecture

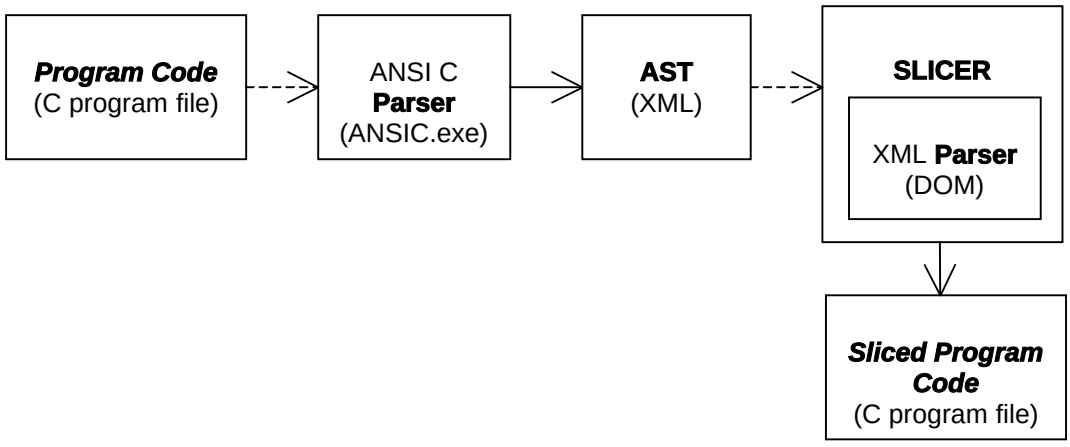

Figure 6: Diagram illustrating the software architecture

Above is a diagram of the software elements (labelled in bold) that are required to slice any program. The dashed lined arrows indicate input to an element, with the arrow leaving the

element inputted to the element it is pointing to. The solid lined arrows indicate that the element it is leaving outputs the component it is pointing to.

The program to be sliced (element 1) is the main input of the slicing process that is entered by the user at the command line as a C program file name. The file is then read into the parser (element 2) whereby it is parsed and transformed into its AST (element 3). After that, the AST is passed to the slicer mechanism (element 4), initiating the slicing process by supplying the slicer with all the necessary information on the program it will slice. Finally, the slicer outputs the sliced program in the form of a C program file.

Looking at figure 6, we see that there are two software element (elements 2 and 4) that are software components, and three software element (elements 1, 3, and 5) that are inputs and outputs to these components.

Our goal is to develop the slicer in the diagram, beginning now with its design.

3.2.2 Modularisation

The slicer functionality is distributed between three classes:

1. Code Converter
2. Code Extractor
3. Slicer Tool

1. Code Converter

The Code Converter class is in charge of converting the raw C code into a CFG graph representation of it. It does this via the ANSI parser by running the parser and capturing the AST XML output in an XML file that Code Converter creates if the c program has no compile-time errors, otherwise, the exception Parser Exception is thrown. This file is then passed to the XML parser DOM so that the AST XML can be traversed. While traversing the AST, the program statements are rebuilt, passing statement data to subclasses of the Syntax class. Control Flow Node classes model CFG nodes, to which Code Converter passes a Syntax class to an instance of. They also are being constructed while traversing the AST. Code Converter calls a method from Code Extractor to compute the token points of the statement stored in the syntax class. When all the Control Flow Node objects have been assembled, Code Converter is able to compute a reverse control flow graph returned by the reverse control flow graph (rcfg) method as a Control Flow Graph data structure object.

2. Code Extractor

Code Extractor is architecturally more straightforward than the Code Converter, in that it is dependent on just one other module, and basically just reads and writes to files, yet it plays a crucial role in the slicer program. Responsible for syntax operations, Code Extractor pinpoints the exact position in the program file of each statement token, passing the line, and begin and end indices to a Token Point data structure object, returning a Token Point object array. The line the statement is on is attained from the AST, and together with the statement code, provide a starting point to locating the begin and end indices of each token.

Another function the Code Extractor performs is creating the file that saves the slice syntax. The position values stored in Token Point objects allow Code Extractor to copy the same syntax found in the original program into the new “slice” file. A collection (array) of Token Point objects of all the statements that the slice should contain are enough to recreate the original program but of course without the unwanted statements (i.e. statements not part of the slice).

3. Slicer Tool

Combining the two components Code Converter and Code Extractor allows us to construct a slice; the Slicer Tool class does just this in addition to placing the slicer’s functionality within a

simple command line user interface environment. All the slicer's inputs and outputs (section 3.1.1) are to be accounted for by the user interface through prompting the user for the specific input, that is, programs to be sliced and slice criteria, and displaying on the command prompt screen the slicer's outputs.

A program file is to be entered at the command prompt as the argument to the Slicer Tool class, which, before becoming the argument to the Code Converter class constructor, is evaluated and validated to confirm the file is sliceable, i.e. is a c program file. If there are no problems with the program file, Slicer Tool will initially ask for the slice criterion, giving an option of program variables to choose from for the slice set, and once entered initiates the slicing process by calling the rcfg method from the Code Converter class and saving the Control Flow Graph object returned in the Slicer Tool's rcfg variable data member.

Computing the slice or executing the slicing algorithm returns a set of node identifiers corresponding to the statement included in the slice. The Control Flow Node objects that have the same node identifiers as those in the slice result are returned from the nodes method of rcfg; node identifiers are used as the nodes array indices to acquire each object. Next we retrieve each Control Flow Node object's Token Point array and put them into one array holding the points of every token of every statement in the slice. Here, the Code Extractor class comes into play, passing all the points to the write code method that creates the file containing the slice, which Slicer Tool then displays for the users viewing.

3.2.3 Algorithm Design

Over all, there are two main algorithms, one for converting the raw C code into a CFG, and the other to compute the slice. An algorithm for converting the code into CFG nodes has not been carefully planned seeing as it is just a matter of traversing the AST using DOM to iterate through the AST nodes extracting all the data relevant to a statement and a CFG node, and adding controlled nodes to their controllers. Likewise, algorithms to deal with extracting the source map of the C code from the original program were not looked into and are beyond the scope of this project. Appendix 2.a has the pseudo code for the two algorithms.

3.2.3.1 Constructing a RCFG

A RCFG is represented as an adjacency digraph, which is a data structure that applies a boolean matrix to map and model nodes and arcs between them. RCFG nodes are the vertices of the graph and the arcs are the edges. A vertex is modelled as a row in the matrix and an edge is a true value or 1 in the column of a vertex. An arc from vertex *a* to vertex *b* is represented as a 1 value at row *a* and column *b*. We can construct a RCFG by simply generating a CFG, whereby an arc from vertex *a* to vertex *b* is represented as a 1 value at row *b* and column *a*, and then transpose the matrix to get the reverse graph. If we go back to section 3.1.2.1, the control structures described have equivalent matrix representation:

1. Sequence - a series of successive nodes from 1 to 6

	1	2	3	4	5	6
1	0	0	0	0	0	0
2	**1**	0	0	0	0	0
3	0	**1**	0	0	0	0
4	0	0	**1**	0	0	0
5	0	0	0	**1**	0	0
6	0	0	0	0	**1**	0

Figure 7: Matrix representation of a sequence of statements

2. Selection\Conditional Statements – a node with a consecutive and alternative branch.

"If" Statement

	1	2	3	4	5	6
1	0	0	0	0	0	0
2	**1**	0	0	0	0	0
3	0	**1**	0	0	0	0
4	**1**	0	**1**	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0

"If Else" Statement

	1	2	3	4	5	6
1	0	0	0	0	0	0
2	**1**	0	0	0	0	0
3	0	**1**	0	0	0	0
4	**1**	0	0	0	0	0
5	0	0	0	**1**	0	0
6	0	0	0	0	**1**	0

Figure 8: Matrix representation of selection statements

3. Iteration statement – a node with node 1 controlling nodes 2 to 6.

	1	2	3	4	5	6
1	0	0	0	0	0	**1**
2	**1**	0	0	0	0	0
3	0	**1**	0	0	0	0
4	0	0	**1**	0	0	0
5	0	0	0	**1**	0	0
6	0	0	0	0	**1**	0

Figure 9: Matrix representation of iteration statements

Sequential statements are just vertices next to each other – a vertex before the previous has an edge to it. In the "if" statement example, the node (vertex) directly after the last node controlled by the "if" node has an edge to the "if" node and the last node(s) the "if" node controls. As for the "else" statement, the first node in both the "if" and "else" branch have an edge to the "if" node and the first "else" branch nodes has no edge to the last "if" branch node. The node after the last "else" branch node has an edge to the last node(s) of the two branches. Loop nodes have an edge to the last node(s) that they control that represent possible control paths from the loop node. Furthermore, the node directly after the last node(s) controlled by loops has an edge to the loop node. Each predicate control structure is dependent on the last node(s) controlled by that structure. Actually attaining the last controlled node(s) of an iteration or selection statement is not difficult; you follow the links between the predicate nodes and their controlled nodes until there are no more nodes left (you reach the last one). The entry and exit nodes are not represented as they do not add any useful information to the adjacency matrix.

3.2.3.2 Computing The Slice

Slicing the RCFG entails traversing it with a loop for vertex iteration, and another for the edges of each vertex. Nodes are processed by vertices inputting their outputs to the CFG nodes that correspond to the vertices' edges. When the current edge of the current vertex is a selection node with an in-degree of two, it uses a stack to hold reference the first of the two inputs to any selection statement. Once we arrive at the second input edge, the reference to the first input is popped of the stack and the union of the two inputs are computed as the input for the selection node. Current edges that are iteration nodes and that have an in-degree of two, indicate an input to them from outside the iteration nodes' statement block. The algorithm, after processing the iteration edge, jumps to the loop node, processes it, and then jumps again but this time to the iteration node's last controlled node.

In the first iteration of the vertices loop, loop nodes edges are monitored and the main (outer most) loop containing the slice node *n* or, if *n* is not controlled by a loop, the closest main loop

to *n* is calculated. Consecutive iterations of the vertices loop iterate from the last statement block (controlled) node of the main loop, identified in the previous iteration, to the first node. The algorithm doesn't iterate more than once if a loop node isn't found before *n*. Iteration ends at the first CFG node when a node outputs the same output, apart from an empty set, more than once.

3.3 Implementation

3.3.1 Code

See appendix b for the source code.

3.3.2 Known Limitations

Not every combination of intra-procedural statements can be tested for. Loops within loops, nested empty predicate statements, and other complex code have unpredictable results, but in most cases, the slicer should be able to cope with slicing the code.

Problem

Just outputting the slice set caused the slice not to contain the slice node in the absence of loops within the program. Including or excluding the slice node identifier as well as the slice set yields inaccurate slices.

Implications for Slicing Result

The slice node identifier is included in the slice result even if it doesn't affect the slice set in any way. No iterations mean that if the slice node identifier isn't outputted with the slice set then the slice node is never part of the slice result.

Current Work-Around

Start the slicing algorithm by processing the slice node with both the slice set and the slice node identifier inputted, and if the slice node doesn't fire then take out the slice node identifier from the slice node's output.

Problem

Code Extractor throws an exception when reading from the file some C code written on the same line.

Implications for Slicing Result

May not compute slice if C program not laid out one line per statement.

Current Work- Around

Non at present.

3.4 Testing

Standard black-box testing will be the approach taken for verifying that the slicer slices programs and constructs precise slices with respect to the slice criterion entered. This approach calls for an input (programs and slice criteria), expected outcome (the slice), and actual outcome (slicer's output).

A randomly selected set of programs and slice criteria for those programs will be presented as input for the slicer to process. Expected results will be compared with the actual result determining whether or not the slicer does what it suppose do, i.e. if the expected outcome is the same as the actual outcome then the slicer works, if not, there is a problem with the slicer.

In the event of an unexpected outcome, the actual outcome is examined and the code modified to achieve the desired result.

3.4.1 Testing Data Space

For the purpose of testing, we will attempt to make the scope of the test data general and simple so as to cover realistic cases where slicing is applied with easy to understand program examples. A limited range of statement configurations that includes a mixture of control structures (section 3.1.2.1) are to be considered for testing; trying to test every possible combination of statements is unfeasible and would take far too long. The slice criteria inputs to the slicer are going to be single and multiple variable identifiers for the slice set at random positions.

The four types of test cases are:

1. sequences
2. selection statements
3. iteration statements
4. combination of the above

3.4.2 Test Result

See appendix 4a for the Test Cases.

4 Conclusion

4.1 Outcome

In this report we have documented the implementation of a miniature slicer program that slices a subset of the C programming language, constructing a backward intra-procedural slice, and displays the slice to the user as detailed in the objective specification (section 2).

A variation on the parallel slicing algorithm approach was adopted to compute slices, and inadvertently lead to the innovation of a new sequential algorithm based upon the old parallel one.

Problems with The Parallel Algorithm

Throughout the project's life-cycle, the parallel algorithm was thoroughly analysed and studied to try and fully understand how it worked. On close inspection, it was observed that the algorithm wasn't optimal in terms of its efficiency as its work increased the effort required to compute the slice. In the presence of loops, the algorithm would iterate and process every node in the CFG for all program instances no matter where their positions were in the program. Thus, nodes\statements after the slice node *n* in the program that didn't lie within the same loop as *n* would also be processed. As the algorithm calculates the static backward slice, this would mean needlessly processing nodes that were irrelevant to the slice with respects to the slice criterion. Moreover, if *n* came before a loop in the program, the algorithm would pointlessly iterate through the whole CFG even if there weren't any loops before *n*.

The New Algorithm

The algorithm introduced in this report is more efficient than the parallel one because it only processes nodes that potentially affect the slice criterion, ignoring all other nodes. Iteration occurs when i) the slice node *n* is contained within a loop, or ii) there is a loop before *n* in the program. These two situations are the only time where CFG iteration is really necessary to check nodes within loops. Furthermore, the node the iteration starts from at each cycle is a significant factor that has been considered to reduce the algorithm's effort. In both the parallel algorithm and the new one, process communication is initiated from *n*, which is specified by the slice criterion. However, the new algorithm restarts the iteration at the last reachable node of either, the main (loop not contained within another) loop that contains *n* or, in the case where *n* doesn't lie within a loop, the closest main loop before *n* in the CFG. If *n* is not contained within a loop then it along with all the nodes before the last reachable node of the closest main loop before *n* are never processed after the initial iteration.

The New Algorithm Steps:

1) Initial iteration - processes all the nodes from the slice node to the first node in the CFG, and checks for loops.
2) If a loop is found then it saves it as the "main loop".
3) If another loop is found then it checks if *main loop* is contained within *this* loop, assigning the main loop with this loop if true.
4) If no loops found then iteration ends after processing first CFG node, else for each consecutive iteration, the loop returns to the main loop's last reachable node processing all nodes from it to the first node in the CFG.

This novel and original change to the parallel algorithm guarantees that efficiency is improved by only iterating through the nodes that may influence and\or be influenced by process communication (i.e. altering the message outputs with new inputs). The following statements prove that the algorithm does what we claim it does, that is, provides an efficiency improvement described above:

Facts:

1) All nodes after the slice node *n* in the program have no affect on *n* unless it is contained within a loop.
2) Nodes not contained within a loop don't receive new input after the first iteration unless they come before a loop in the CFG; their output remains the same.
3) Because loops have arcs to the last statement(s) that are controlled by them, a loop's output is inputted to all its last statement(s) causing a process communication cycle that continues until a label has the same output more than once.

4.2 Interpretation of Results

On the whole, the majority of the objectives were accomplished to a satisfactory standard, and all were completed to some reasonable degree. The work produced on the slicing algorithm was original and hopefully made the project more note worthy.

4.3 Future Work

An obvious extension to this project would be to enlarge the subset of the C language to perhaps allow derived variables, enumerations, and inter-procedural slicing using the same CFG algorithm. Of course, the algorithm would have to be extended to handle procedural properties and might call for the CFG to be transformed to facilitate these properties. Another suggestion would be to slice another language, like Java or C++, and attempt to slice different object oriented features. Although, the efficiency of the parallel algorithm has been improved, it hasn't been proven to be work or speed optimal. A natural extension would be to optimize the algorithm increasing its efficiency and\or speed, or measuring these algorithmic properties to decide whether or not it is optimal. Slicing with another approach, for example using PDGs and\or Data-Flow equations, and comparing approaches is another suggestion for extending the project. Slicing programs in a graphical user interface environment would present the slice in a fashion more suitable for reading code.

References

[Abr84] Abramsky, S., *Reasoning about Concurrent Systems in Distributed Computing*, University of London, pp.307-319, 1984.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Ban79] J.P. Banning. *An efficient way to find the side effects of procedure calls and the aliases of variables*. In Conference Record of the Sixth ACM Symposium on Principles of Programming Languages, pages 29–41, 1979.

[Bin93a] David Binkley. *Slicing in the presence of parameter aliasing*. In *Proceedings of the Third Software Engineering Research Forum*, pages 261–268, Orlando, Florida, November 1993.

[Bin93b] David Binkley. *Precise executable interprocedural slices*. ACM Letters on Programming Languages and Systems, 2(1–4):31–45, 1993.

[BG96] David Binkley and Keith Brian Gallagher. *Program slicing*. Advances in Computers, 43:1-50, 1996. Pages 1 – 2.

[BH+00] D. W. Binkley, M. Harman, L. R. Raszewski and C. Smith. *An empirical study of amorphous slicing as a program comprehension support tool.* 8th IEEE International Workshop on Program Comprehension (IWPC 2000), Limerick, Ireland, June, 2000. Pages 161-170.

[BE93] Jon Beck, David Eichmann: *Program and Interface Slicing for Reverse Engineering,* In Proceedings of Working Conference on Reverse Engineering and the International Conference on Software Engineering, Baltimore, MD, May 1993, pp. 509-518.

[CC93] T. Y. Chen and Y. Y. Cheung. *Dynamic Program Dicing.* IEEE Conference on Software Maintenance (ICSM 1993), pp. 378-385, 1993.

[CCD98] G. Canfora, A. Cimitile and A. De Lucia. *Conditioned program slicing.* Information and Software Technology special issue on Program Slicing, 40(11-12), pages 595-607, December 1998.

[CDM96] A. Cimitile, A. De Lucia and M. Munro. *A Specification Driven Slicing Process for Identifying Reusable Functions.* Software Maintenance: Research and Practice, 8, pages 145-178, 1996.

[DF+00] S. Danicic, C. J. Fox, M. Harman and R. M. Hierons. *ConSIT: A Conditioned Program Slicer.* IEEE International Conference on Software Maintenance (ICSM 2000), San Jose, California, USA, October, 2000. pages 216-226.

[DHS95] Sebastian Danicic, Mark Harman and Yoga Sivagurunathan. *A Parallel Algorithm for Static Program Slicing*. Information Processing letters, 56(6):307-313, 1995.

[FOW87] J. Ferrante, K.J. Ottenstein, and J.D. Warren. *The program dependence graph and its use in optimization*. ACM Transactions on Programming Languages and Systems, 9(3):319–349, 1987.

[FRT96] J. Field and G. Ramalingam and F. Tip. *Parametric Program Slicing.* 22nd Symposium on Principles of Programming Languages (POPL'95), January 22-25, 1995, pages 379-392.

[Gal90] Keith Brian Gallagher. *Using Program Slicing in Software Maintenance.* Ph.D. Thesis, University of Maryland Baltimore County, 1990

[Hec77] Hecht, M. S. *Flow Analysis of Computer Programs*. Elsevier, 1977.

[Hoa85] Hoare, C. A. R., *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[HD95] Mark Harman and Sebastian Danicic. *Using Program Slicing to Simplify Testing*. Journal of Software Testing, Verification and Reliability, 5(3):143-162, 1995.

[HD97] Mark Harman and Sebastian Danicic. *Amorphous Program Slicing.* IEEE International Workshop on Program Comprehension (IWPC'97), Dearborn, Michigan, May 1997, pages 70-79.

[HH01] Mark Harman and Rob Hierons. *An Overview of Program Slicing.* Software Focus. 2(3):85-92, 2001.

[HPR88a] Susan Horwitz, Jan Prins, and Thomas Reps. *Integrating non-interfering versions of programs*. In Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, pages 133–145, 1988.

[HPR88b] S. Horwitz, J. Prins and T. Reps. On *the adequacy of program dependence graphs representing programs*. Proc. 15th ACM Symposium on Principles of Programming Languages, 1988, pages 146-157.

[HRB88] Horwitz, S., Reps, T. and Binkley, Inter procedural slicing using dependence graphs, Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation Atlanta, GA (1988), pp.25-46; also in: SIGPLAN Notices Vol.23, pp.35-46, 1988.

[HR92] S. Horwitz and T. Reps. *The use of program dependence graphs in software Engineering*, Proc. 14th Internat. Conf. On Software Engineering, Melbourne, Australia, pp.392-411, 1992.

[Jin93] J. Cheng. *Slicing concurrent programs - a graph-theoretical approach*. In Automated and Algorithmic Debugging, volume 749 of Lecture Notes in Computer Science, pages 223-240. Springer-Verlag, 1993.

[Jia99] J. Zhao. *Slicing concurrent Java programs*. In Proceedings of the 7th IEEE International Workshop on Program Comprehension, pages 126-133, May 1999.

[JH99] J. Ahn and T. Han, *Static Slicing of a First-Order Functional Language based on Operational Semantics*, Korea Advanced Institute of Science & Technology (KAIST) Technical Report CS/TR-99-144, Dec., 1999.

[JR94] D. Jackson and E. J. Rollins. *Chopping: A Generalization of Slicing.* Technical Report, Carnegie Mellon University, School of Computer Science, Number CS-94-169, p. 21, July 1994.

[KK+81] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth ACMSymposium on Principles of Programming Languages*, pages 207–218, 1981.

[KNN99] G. Kókai and J. Nilson and C. Niss. *GIDTS - A Graphical Programming Environment for Prolog.* ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE), pages 95-104, 1999.

[KR98] B. Korel and J. Rilling. *Dynamic program slicing methods.* Information and Software Technology special issue on Program Slicing, 40(11-12), pages 647-659, December 1998.

[LH96] L. D. Larsen and M. J. Harrold. *Slicing object-oriented software.* 18th International Conference on Software Engineering (ICSE), pages 495-505, Berlin, March 1996.

[LW86] J. R. Lyle and M. Weiser. *Experiments on Slicing-Based Debugging Tools.* Empirical Studies of Programming, Elliot Soloway (editor), Ablex Publishing, Norwood, New Jersey, June 1986.

[Mic94] Michael Ernst. *Practical fine-grained static slicing of optimized code*. Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA, 1994.

[OO84] Ottenstein K. J. and Ottenstein, L. M., *The program dependence graph in software development environments*, SIGPLAN Notices, Vol.19, pp.177-184, 1984.

[OB98] L. M. Ott and J. M. Bieman. *Program Slices as an Abstraction for Cohesion Measurement.* Information and Software Technology special issue on Program Slicing, 40, (11-12), pages 691-700, November 1998.

[OT89] L.M. Ott and J. Thuss. *The relationship between slices and module cohesion*. In Proceedings of the 11th International Conference on Software Engineering (1989), pp. 198-204.

[RHS94] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. *Speeding up slicing*. In Proceedings of the Second ACM SIGSOFT Conference on Foundations of Software Engineering, New Orleans, LA, December 1994. To appear.

[RSH94] T. Reps, M. Sagiv, and S. Horwitz. *Interprocedural dataflow analysis via graph reachability*. Report DIKU TR 94-14, University of Copenhagen, Copenhagen, 1994.

[Ste98] C. Steindl. *Intermodular Slicing of Object-Oriented Programs.* International Conference on Compiler Construction, Lecture Notes in Computer Science, LNCS, Vol. 1383, pages 264-278, 1998.

[SD96] S. Schoenig and M. Ducasse. *A Backward Slicing Algorithm for Prolog.* Lecture Notes in Computer Science, Vol. 1145, p. 317-326, 1996.

[Tip95] Frank Tip. *A survey of program slicing techniques*. Journal of programming languages, 3(3), September 1995.

[Tip96] F. Tip, J.-D. Choi, J. Field and G. Ramalingam, S*licing Class Hierarchies in C++.* SIGPLAN Notices, Vol. 31, 10, pp. 179-197, ACM Press, October 6-10 1996.

[Wei79a] Mark Weiser. *Experiments on Slicing-Based Debugging Aids.* in Empirical Studies of Programmers, pp. 187-197, Ablex Publishing Corporation, 1986.

[Wei79b] Mark Weiser. *Program slices: Formal, psychological and practical investigations of an automatic program abstraction method*, Ph. D. Thesis, University of Michigan, Ann Arbor, MI, 1979.

[Wei82] Mark Weiser. *Programmers Use Slicing When Debugging.* Communications of the ACM, Vol. 25(7), pp. 446-452, July 1982.

[Wei84] Mark Weiser. *Program Slicing*. IEEE Transactions on Software Engineering, vol. 10, no. 4, July 1984, pp. 352-357.

[WA98] M. R. Woodward and S. P. Allen. *Slicing Algebraic Specifications*, Information and Software Technology, ISSN: 0950-5849, 40(2), 105-118, May 1998.

[ZCU96] J. Zhao, J. Cheng, and K. Ushijima. *Static slicing of concurrent object-oriented programs.* Proceedings of 20th International Conference on Computer Software and Applications, Seoul, Korea, IEEE CS Press, 1996, pages 312-320.

[ZCU97] J. Zhao, J. Cheng, and K. Ushijima. *Slicing concurrent logic programs.* In T. Ida, A. Ohori, and M. Takeichi, editors, Second Fuji International Workshop on Functional and Logic Programming, pages 143-162, 1997.