

Integration of generic program analysis tools into a software development environment

Erica Glynn

Ian Hayes

Anthony MacDonald

School of Information Technology and Electrical Engineering
University of Queensland
email: {erica, ianh, anti} @itee.uq.edu.au

Abstract

Support for program understanding in development and maintenance tasks can be facilitated by program analysis techniques. The flow analysis techniques of control-flow and data-flow analysis support program comprehension through the provision of graphical and interactive representations.

This paper outlines the addition of program analysis support to a generic language-based environment via the construction of the automatically generated Graph Builder, and hand-crafted generic Analysis tool components. These generically implemented tools represent a significant advancement in integrated program understanding support within a software development environment and the integration of these tools represents a step towards an ideal language support environment as definitively presented by the Stoneman requirements specification.

The realisation of flow analysis support tools for program understanding further requires the development of new techniques for calculating execution characteristics and modification-use relationships of control-flow graph nodes within the context of a generic environment supporting relational navigation.

1 Introduction

Program comprehension is the process of acquiring knowledge about a computer program. Increased knowledge enables such activities as bug correction, enhancement, reuse, and documentation. While efforts are underway to automate the understanding process, such significant amounts of knowledge and analytical power are required that today program comprehension is largely a manual task.

– Rugaber (Rugaber 1995)

The use of program analysis techniques, including control-flow and data-flow analysis, has been identified as supporting software understanding and comprehension tasks (Hecht 1977). Program analysis tools can reduce time and resources required throughout the development process by supporting comprehension and thus decrease the costs involved.

Generic and integrated tool-sets are used to reduce costs associated with the development of software by aiding developers through the provision of common

usage paradigms for tools across programming languages, and for the various languages used by those tools. Providing adequate support for efficient utilisation of tools is a major goal of the software development environment research area (Reiss 1990). An ideal software development environment provides an integrated set of tools for use throughout all stages of the software life cycle (Howden 1982, Reiss 1990). The integration of program analysis tools into software development environments will bring about a higher level of support for development engineers undertaking program understanding tasks.

The Stoneman (Howden 1982, Web 1980) requirements specification for software development environments, describes static program analysis tools as a necessary part of a minimal language support environment. In this context, static program analysis tools provide support to developers by assisting with program understanding tasks. A recent informal study conducted by the authors, evaluated contemporary software development environments using the requirements defined by Stoneman, finding that support for program understanding through the use of integrated static flow analysis tools is still not well provided within modern, commercial, integrated development environments.

Vital to the development of any software project is the environment with which it is created and managed. Software development environments can range from text editors used in conjunction with command line compilers to enhanced integrated language-based environments giving in-line context sensitive help, presentation and incremental compilation. In this project the generic, integrated, software development environment UQ* (Jarrott & MacDonald 2003, Toleman, Carrington, Cook, Coyle, Jones, MacDonald & Welsh 2001) was extended with the addition of flow analysis tools to support program understanding tasks.

Throughout compilers literature, algorithms for the implementation of flow analysis are thoroughly presented. During this project, it was found that the techniques presented by these resources were at times overly complex, tightly coupled to other techniques and not suitable within a program understanding context due to the use of intermediate and low level machine code representations. Accordingly in this project, the development of two new techniques for the computation of flow analysis information operating on actual code artefacts within the context of a integrated, generic development environment was required.

Genericity creates a significant problem for control-flow analysis. In order to perform control-flow analysis, the control constructs of the program need to be identifiable. By the use of an automatically constructed tool and the Prolog-like attribute grammar language of (Cook, Welsh & Hayes 2002), a

solution in fitting with the generic nature UQ \star was generated.

Section 2.1 introduces program analysis and the concepts that impact on the ability to integrate existing program analysis techniques into a programming support environment. A brief overview of the programming support environment (UQ \star) used within this project is given in Section 2.2. The technical aspects of the tools developed are discussed in Sections 4 and 5. The paper closes with a discussion of the outcomes of this work

2 Background

2.1 Program analysis

Control-flow analysis is the study of the control structures of a program. Best explained through diagrams, the transformation of source code to control-flow graph is visible in Figure 1. In this example, a program written in PL0 (Wirth 1976) is undergoing transformation. In addition to the derivation of control-flow graphs, control-flow analysis provides cycle identification as described in (Muchnick 1997). The developed control-flow graphs and cycle calculations can be further used in more advanced program analysis techniques including data-flow analysis.

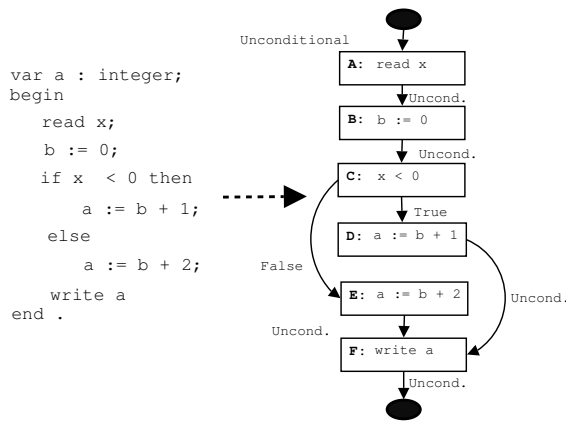


Figure 1: Example program and control-flow graph

Data-flow analysis represents a suite of techniques that study how a program utilises data values. One component technique of the data-flow analysis suite is modification-use. Modification-use relations are the point-to-point mappings of the use of identifiers to the statement or set of statements that potentially last modified their value. By chaining together the point-to-point mappings, modification-use can provide a valuable means of statically determining the data dependencies for a given variable. Figure 2 shows the modification-use mappings created for the program and control-flow graph of Figure 1. Visible in the diagram are both the point-to-point mappings of identifiers (e.g., x) to modifying statements, and a chain of such mappings (e.g., a at **F** depends on b at **B**).

2.2 Software development environments

Work at The University of Queensland has produced an experimental software development environment, UQ \star (Toleman et al. 2001, Welsh, Broom & Kiong 1991), which is distinguished by its facilities for capturing and manipulating relational structures within and between software documents, and for displaying these structures in textual or diagrammatic form (Jones & Welsh 1997, Jarrott & MacDonald 2003). UQ \star is a generic language-based

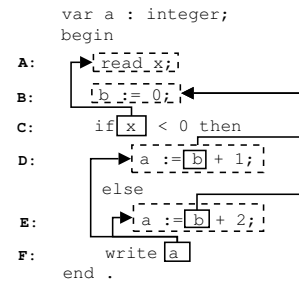


Figure 2: Example modification-use chain

environment in that, when provided with appropriate descriptions of the languages and representations (Toleman et al. 2001) involved in a development method, it provides its users with language-specific support. The UQ \star architecture is shown in Figure 3. A central document server manages the syntactic and relational structures and ensures their persistent storage (MacDonald & Welsh 1999) as appropriate. Syntactic structures are represented as abstract syntax trees, while N-ary relational structures are non-hierarchical connections within and between elements of software documents (as shown in Figure 4) and between software documents and data. Software documents in this context are both abstract syntax trees and relations. The document server makes these structures accessible to a collection of tools (Welsh & Yang 1994). These tools can be categorised depending on whether they *interact* with the user or not, and whether they *construct* syntactic and relational structures or not. A language-based editor is an example of a user-interactive and constructive tool (Cook & Welsh 2001). A static semantic analyser is an backend tool that generates relational information linking syntactic constructs of its input document with error messages and is thus considered an example of a non-user-interactive, constructive tool.

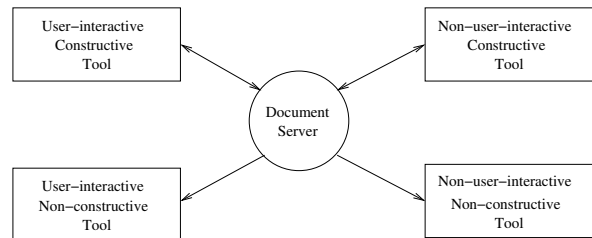


Figure 3: UQ \star architecture

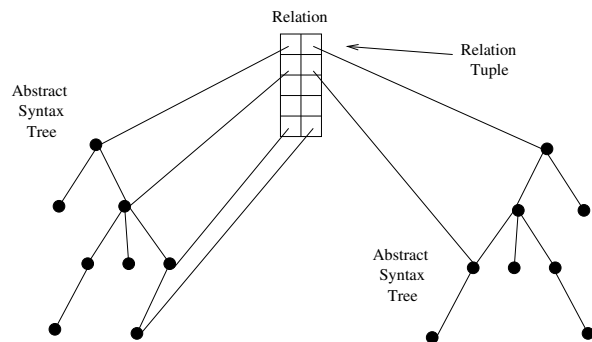


Figure 4: UQ \star syntactic structures

In general, UQ \star relations are N-ary, bi-directional,

and do not necessarily involve only viewable document locations. The elements in relation tuples can be strings or other attributes in addition to references to document components. The underlying model of UQ \star relations is thus more akin to that used in relational databases.

A recent addition to the UQ \star programming support environment is the automatic specification of relation building tools via the use of a meta-language (Cook et al. 2002) extending the existing environment description language. Based on an attribute grammar style syntax (Aho, Sethi & Ullman 1986), the meta-language can be described as a Prolog-style language utilising recursion to implement potentially complex abstract syntax tree traversals. Tools created in this manner can both *use* existing and *provide* new relations and abstract syntax trees to the system for utilisation in other components.

3 Design

In this research, a system was developed to conduct generic flow analysis. For simplicity, this paper focuses on the flow analyses as they apply to the language PL0 (Wirth 1976). To allow the generic computation of flow analysis, two UQ \star tools were utilised. The two tools developed are the Graph Builder (Section 4) and the Analysis tool (Section 5). Using the tool classifications of Section 2.2, both of these tools can be categorised as on-user-interactive and constructive. The use of *global* (stored in the UQ \star document server) relations allows the collaboration of the two tools, in addition to making the results of the flow-analyses available for use by other UQ \star tools. This capability will allow techniques that rely on the results of control-flow analysis, such as Timing-Constraint Analysis (Grundon, Hayes & Fidge 1997), to be implemented in UQ \star in future projects.

The first of the tools is the language-specific Graph Builder. The Graph Builder is written using UQ \star 's EDL attribute grammar language (Cook et al. 2002). The main task of the Graph Builder is to populate the relations which define the control-flow graph of a program. The attribute grammar additionally extracts language specific information such as concrete syntax (needed for results presentation) and abstract syntax (for language semantics such as what non-terminals can modify and/or use the values of variables) needed for use in the second tool, the generic Analysis tool. Through the use of the attribute grammar, it is the Graph Builder component that determines what control structures (sequence, selection and iteration) exist for a given language. An example of such, is that in PL0, “**while** **Condition** **do** **Statement**” represents an iterative construct that will execute the **Statement** until the **Condition**'s value becomes false.

The second of the tools is the generic, language-independent Analysis tool. Concerned with creating call-graphs, conducting data-flow analysis (to determine the modification-use relationships) and the presentation of the results of both styles of flow analysis, the Analysis tool uses the relations populated by the Graph Builder to generically perform its tasks. The amount of effort required to set up flow analysis for additional languages was minimised by making as much of the system generic as possible. The reusability of the Analysis tool component derives from its generic nature. Additionally, the need for genericity is directly responsible for the decision not to implement more of the data-flow analysis in the EDL attribute grammar.

The tools of this research were developed for the UQ \star generic, integrated development environment.

From an integrated development environment perspective, it is important that the developed tools placed minimal requirements on the system that they are to work with. This includes conforming to existing usage paradigms for UQ \star . This requirement dictated that the tools would be generic and incremental where possible. It was the need to work within the existing UQ \star environment that motivated the creation of the two tool approach. By making the Analysis tool language-independent using generic data-flow analysis techniques, the amount of the system that requires re-definition for each subsequent language is minimised. Had a tool been developed from first principles to compute the flow analysis for PL0 only, each additional language would require the complete re-development and/or porting of the tool, rather than just the attribute grammar of the language-specific Graph Builder.

Placing minimal requirements on the system being integrated with also includes using existing language definitions, even where such definitions are sub-optimal. For example, the flow analysis tool developed in this research works with the PL0 (Wirth 1976) language variant defined for use in UQ \star . The full definition of this language variant appears in (Toleman, Carrington, Cook, MacDonald & Welsh 1999). Unfortunately the design of the language is not optimal for conducting data-flow analysis, especially using a generic approach. Rather than require changes to the language, which could potentially impact on existing non-generic tools developed for UQ \star 's PL0 language, the appropriate direction for this research project was to develop methods that were flexible enough to work with existing language designs. An example of the need to modify an approach to account for esoteric language designs can be seen with the data-flow relations described in Section 4.3.1.

4 Graph Builder

The Graph Builder tool was defined using a Prolog-like attribute grammar language (Cook et al. 2002) and compiled into an executable binary (via C++ code) using the EDL compiler. It is the Graph Builder that is responsible for the creation and population of the relations representing a program's control-flow graph. This language-specific tool is designed to operate incrementally to extract all information needed to enable control-flow and data-flow analysis to be conducted in a language-independent manner within the Analysis tool (Section 5). The Graph Builder interacts within the UQ \star system in a constructive, non-user-interactive context, creating and populating relations but not interacting with the user or any other tools except through the interfaces provided by the document server.

The relations of the Graph Builder are used to model the control-flow graph of a given program as well as extract required information for use in the language-independent Analysis tool.

Within the Graph Builder tool there exists two classes of relations: those occurring only within the `flowAnalysis` attribute grammar, known as *local* or *attribute* relations, and those populated by the Graph Builder tool, available for use throughout the UQ \star system, known as *global* or *document server* relations.

4.1 Control-flow

The control-flow relations of the Graph Builder define a statement based control-flow graph for a supplied program. We represent a control-flow graph via three relations: *GraphEntry* for modelling the root of the

graph, *GraphTrans* for the internal flow edges, and *GraphExit* representing the final nodes in the graph. As shown in Figure 5, it is the combination of these relations that provide the complete control-flow graph.

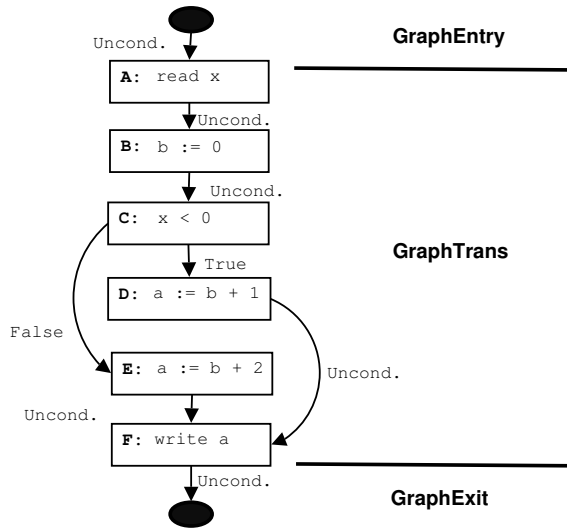


Figure 5: Control-flow graph relations

Representing the internal edges of the control-flow graph, the *GraphTrans* relation is the fundamental component of the Graph Builder’s output. The *GraphTrans* relation is defined as mapping one node to its successor with an labelled edge. In the EDL, *GraphTrans* is declared as `Relation GraphTrans (Node, Label, Node)`. Within the current configuration the three edge labels are *Unconditional* for edges that must be taken, and *True* and *False* representing conditional edges from branching constructs such as those found in `if`, `while` and `repeat` statements. The labels were purposely defined to allow extensions for unrestricted branching including the multiple branching `case` conditional, and arbitrary jumping `goto` statements. Figure 5 presents an example of labelled arcs.

Using the sample program of Figure 1, the contents of the *GraphTrans* relation after the incremental invocation of the Graph Builder are displayed in Table 1.

Node (from)	Label	Node (to)
A	<i>Unconditional</i>	B
B	<i>Unconditional</i>	C
C	<i>True</i>	D
D	<i>False</i>	E
E	<i>Unconditional</i>	F
F	<i>Unconditional</i>	G

Table 1: *GraphTrans* relation for the example program.

The *GraphExit* relation stores the nodes for the end(s) of a control-flow graph, and is defined as:

`relation GraphExit (Node, Label).`

Within the Analysis tool, the *GraphExit* relation is used during the abstract syntax tree traversal to determine when the end of a flow graph is reached. Table 2 shows the contents of the *GraphExit* relation for the program of Figure 1.

Additionally, a control-flow graph may possess multiple exit points. Multiple exits may occur where a conditional statement is the last statement of a program/procedure, and in languages such as Java

Node	Label
F	<i>Unconditional</i>

Table 2: *GraphExit* relation for the example program.

, where a method can have multiple `return` statements. Figure 6 provides an example of program with two exit points. Table 3 shows the contents of the *GraphExit* relation for this example.

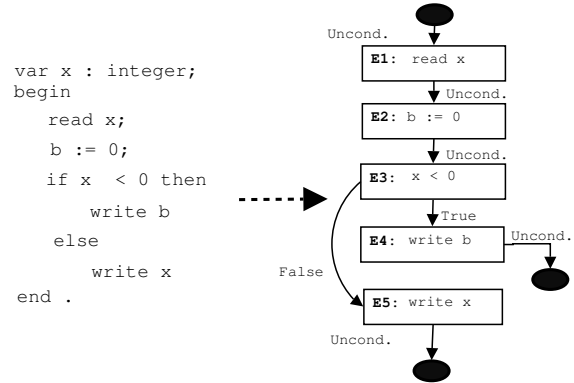


Figure 6: Program with multiple exits

Node	Label
E4	<i>Unconditional</i>
E5	<i>Unconditional</i>

Table 3: *GraphExit* relation for the program with multiple exits.

The *GraphEntry* relation, defined as `relation GraphEntry (Label, Node)`, holds the node for the beginning of a control-flow graph. As with the *GraphTrans* and *GraphExit* relations, the tuples of the *GraphEntry* relation represent a labelled edge, however unlike the other control-flow relations, the label must be *Unconditional* and each flow graph possesses only one entry. The *GraphEntry* label may seem redundant, but is possible that the supporting of multiple-branching statements (e.g., `case` statements) may require values other than *Unconditional*. The label also remains a component of the *GraphEntry* relation to provide a degree of consistency between the *GraphTrans*, *GraphExit* and *GraphEntry* relations. Table 4 shows the contents of the *GraphEntry* relation for the example program of Figure 1.

Label	Node
<i>Unconditional</i>	A

Table 4: *GraphEntry* relation for the example program.

4.1.1 Procedures

The Graph Builder maintains two structurally equivalent sets of control-flow relations corresponding to collections for a program’s main body (*GraphEntry*, *GraphExit* and *GraphTrans*) and its procedures (*GraphProcEntry*, *GraphProcExit* and *GraphProcTrans*). Issues can arise from conducting analysis in the presence of procedures. Inter-procedural analyses are complicated by:

- analysing across Abstract Syntax Trees (ASTs) (Louden 1997),
- parameters, aliasing, variable scope and scope precedence (Tip 1995),
- the mismatch between the static flow analysis techniques and dynamic nature of recursion,
- how procedure bodies are represented in program’s control-flow graph, and
- the inability to determine the origin (main program or procedure) of tree nodes in a relation.

These issues provided the motivation for the separation of the two relation sets. Furthermore, it was decided that by making the structure of the procedure and main method relations consistent, a single tree traversal algorithm could be constructed for the Analysis tool (Section 5).

In addition to the *GraphProcEntry*, *GraphProcExit* and *GraphProcTrans* relations, the Analysis tool requires the mapping of the name of each procedure to the nodes representing the entry and exit points of its subgraph. This information is used in the construction of call-graphs, and during the application of data-flow analysis to globally scoped variables used within a procedure. The relation declared to hold this information is the *GraphProcMap* relation. This relation is defined by the EDL

```
relation GraphProcMap (String, Node, Node).
This corresponds to the name of the procedure, its entry and exit nodes. It is a constraint of the system that the entry and exit nodes within GraphProcMap must exist within the GraphProcEntry and GraphProcExit relations.
```

4.2 Relation Population

The Graph Builder tool is automatically generated from the compilation of the `GraphBuilder.edl` file. The file defines a Prolog-like attribute grammar used to calculate the control-flow graph of a program, and identify required semantic information specific to the language the program is written in. As the EDL file provides the language-specific information, it is the the portion of the flow analysis system that requires re-definition for each additional language. The file provides *phylums* which defines the inherited and synthesised attributes for each non-terminal, and the *rules* that specify the actual attribute grammar. The semantics of the EDL require that for every non-terminal defined a language, a *phylum* must be declared. Additionally, for every option on the right-hand side of the language’s EBNF declaration a *rule* is defined which populates the attributes declared in the associated *phylum* declaration. It is the syntax used within the *rules* of the EDL attribute grammar language that bear the greatest resemblance to Prolog. Figure 7 provides the example of the `Statement` non-terminal from the PL0 language defined in (Toleman et al. 2001).

The attributes used to calculate the control-flow graph in the EDL are the relations *entry*, *trans* and *exit*. These relations are structurally equivalent to the *GraphEntry*, *GraphTrans* and *GraphExit* relations previously discussed. The attribute relations have been distinguished from the graph relations to aid readability and understandability of the system.

Three clear classes of constructs can be identified in the attribute grammar based upon the *entry*, *trans* and *exit* relations. The classes of constructs are: *Primitive* (or *Atomic*) such as an assignment or condition, *Sequential* such as that of a statement list, and *Branching* which include conditional and looping

```
Statement = ReadStatement | IfStatement
          | ... .
```

(a) Language EDL

```
phylum Statement
{
attributes:
    relation entry (Label, Node) .
    relation exit (Node, Label) .
    relation trans (Node, Label, Node) .
defaults:
    this->entry = empty .
    this->exit = empty .
    this->trans = empty .
}

rule Statement = ReadStatement
{
...
}

rule Statement = IfStatement
{
...
}
...

```

(b) Attribute grammar EDL

Figure 7: Language, phylum and rule definition for `Statement`

constructs such as `if` and `while`. It is the sequential and branching statements that define the control-flow graph of a program.

Primitive constructs are characterised by having no control-flow transitions (an empty *trans* relation), and are usually the entry or exit of a compound construct. For example, a condition (`x < 0`) is an atomic structure that provides the entry to a branching `if-else` statement. Figure 8 is the EDL rule specification of a `Condition`. From the *entry* and *exit* defined in the `Condition` rule, the `if-else` rule of Figure 9 can assert that the entry to the statement as a whole (i.e. the construct that is first executed) is the condition. The rule also demonstrates how the transitions in the control flow-graph are built up using the entry and exit points of the lower level constructs. In this case the rule asserts that there are transitions from the condition to the nodes that form the entry points of the component statements (`s1` and `s2`), and that these happen on the two cases of *true* and *false*. The two potential exits of an `if-else` statement are identified as the exits of each of the statements (`s1` and `s2`). Finally, the rule completes the attribute grammar by adding all nested transitions from `s1` and `s2` to its own set of transitions. Figure 10 presents the `if-else` statement from the example program of Figure 1, with the labelled graph edges classified as the attribute relation they represent.

Rule definitions for sequential constructs are similar those of the branching constructs. The transitions for the sequences are built up using the entry and exits of the component constructs, in the same manner as for branching constructs. For example, a list of statements is linked together by adding a transition from the node representing the exit of one statement to the node for the entry of the next.

The population of the global relations (i.e., *GraphEntry*) can also be found in the flow analysis EDL definition. The population of these relations is demonstrated in Figure 8 with the addition of a tuple to the global *UseTokens* relation (explained in Section 4.3.1), signifying that a condition can use the value of a variable at any point within the construct. It is

```

rule Condition = Exp
{
  this->entry(UNCOND, this).
  this->exit(this, UNCOND).
  this->trans = empty.

  Relations::UseTokens("PL0", "Condition",
    null, null) .
}

```

Figure 8: Specification for Condition primitive construct.

```

rule IfStatement = "if" Condition "then"
  < s1: Statement >
  "else" < s2: Statement >
{
  forall(Node n)
    this->entry(UNCOND, n)
    :- Condition->entry(UNCOND, n) .

  forall(Node n, Label lbl)
    this->exit(n, lbl)
    :- s1->exit(n, lbl) .
  forall(Node n, Label lbl)
    this->exit(n, lbl)
    :- s2->exit(n, lbl) .

  forall(Node n1, Node n2)
    this->trans(n1, TRUE, n2)
    :- Condition->exit(n1, UNCOND) ,
    s1->entry(UNCOND, n2) .
  forall(Node n1, Node n2)
    this->trans(n1, FALSE, n2)
    :- Condition->exit(n1, UNCOND) ,
    s2->entry(UNCOND, n2) .

  forall (Node n1, Label lbl1, Node n2)
    this->trans(n1, lbl1, n2)
    :- s1->trans(n1, lbl1, n2) .
  forall (Node n1, Label lbl1, Node n2)
    this->trans(n1, lbl1, n2)
    :- s2->trans(n1, lbl1, n2) .
}

```

Figure 9: if-else statement rule definition

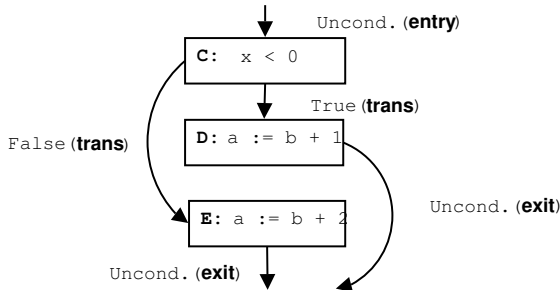


Figure 10: if-else attribute grammar relations

the information held in these relations that enables the generic approach to data-flow analysis developed by the Analysis tool to be possible.

4.3 Language Semantics

Another class of global Graph Builder relations are those that capture the implicit semantics of a given language for use in the generic Analysis tool. These relations hold the terminal and non-terminal symbols that allow language-independent application of control-flow and data-flow analysis techniques. The inclusion of a string element representing the name of the source language in each of the language semantics relations allows the analysis calculation tool to be generic. Moreover, the use of the language name, which is coded into the attribute grammar, effectively means that flow analysis can be conducted by the tool on programs from a diverse range of languages in one

development session.

4.3.1 Data-flow relations

The first of the language semantics relations are the *ModTokens*, *UseTokens* and *BothTokens* relations. These relations capture the language constructs (specifically the non-terminals of the language declared by the appropriate EDL definition) that can change and/or use the value of a variable.

The *ModTokens*, *UseTokens* and *BothTokens* relations are structurally equivalent, with tuples taking the form of (String, String, String, String). As previously identified, the first element is the name of the language to which this semantic information applies. The second element is the name of the non-terminal construct that contains the modification and/or use. The third and fourth elements specify symbols from the language's concrete syntax that are used to denote the start and end of the modification and/or use. By using the concrete syntax and the information stored in the *IdentTokens* relation (discussed in Section 4.3.2), the Analysis tool's data-flow analysis calculations can record the modifications and/or uses for each particular statement instance. Moreover, by using the start/end concrete symbols, constructs can modify and use multiple identifiers. For example, given the statement $a := b + c$, the value of a is modified and the values of both b and c are used.

Another construct that provides multiple modifications and uses is the *multiple assignment* statement (Gries 1987). For example the statement $a, b := c, d$ (which is equivalent to $a := c; b := d$ provided none of a, b, c and d are aliases or pointers), modifies the values of a and b , and uses the values of c and d . Additionally, by using the existing concrete syntax, a given language need no revision or re-definition to be eligible for the flow analysis techniques of this system. Tables 5 and 6 show the contents of the *ModTokens* and *UseTokens* relations for PL0 restricted to those relations of Figure 1. The following relation examples will also be restricted to only those constructs relevant to the example.

String (language)	String (non-terminal)	String (start)	String (end)
PL0	ReadStatement	read	;
PL0	AssignStatement	null	:=

Table 5: *ModTokens* relation for the example program.

String (language)	String (non-terminal)	String (start)	String (end)
PL0	WriteStatement	write	;
PL0	AssignStatement	:=	;
PL0	Condition	null	null

Table 6: *UseTokens* relation for the example program.

Tables 5 and 6 also show the use of `null` where no start or end point of the modification/use exists. For example, the `null` in the second line of Table 5 can be interpreted as meaning the modification(s) in an `AssignStatement` begin at the beginning of the construct.

The *BothTokens* relation is used for language constructs that both modify and use a variable using a single non-terminal. Often representing language short-hands, an example of a statement that both

modifies and uses a variable is the `i++` and `i += 10` syntax from C.

4.3.2 EDL Symbol relations

The other language semantics relations are the *IdentToken* and *ProcCallToken* relations. Both defined as $(\text{String}, \text{String})$, these relations hold the semantic information needed to determine the lexeme symbol that represents an Identifier (i.e., a constant or variable), and the non-terminal symbol for a procedure call, respectively. The information stored by the *IdentTokens* relation is used by the generic Analysis tool in conjunction with the *ModTokens*, *UseTokens* and *BothTokens* relations to conduct data-flow analysis. The *ProcCallTokens* relation, also used in the generic analysis algorithms of the Analysis tool, provides the information required to construct the call-graphs required for the data-flow analysis of globally scoped variables. For the PL0 program of Figure 1, the relevant contents of the *IdentTokens* and *ProcCallTokens* relations are shown in Tables 7 and 8 respectively.

String (language)	String (lexeme)
PL0	ident

Table 7: *IdentTokens* relation for the example program.

String (language)	String (non-terminal)
PL0	ProcCall

Table 8: *ProcCallTokens* relation for the example program.

4.4 Generic Presentation

The generic display relations, *DisplayVarTokens* and *DisplayFixedTokens*, hold the terminal and non-terminal symbols that allow generic presentation of results of control-flow and data-flow analysis. The *DisplayVarTokens* relation, defined by:

relation `DisplayVarTokens (String, String)`. This relation provides the lexeme tokens (second element) for a language (first element) that are to be printed. Examples of lexemes that are printed by the Analysis tool can be seen in Table 9.

String (language)	String (lexeme)
PL0	ident
PL0	number

Table 9: *DisplayVarTokens* relation for the example program.

Similarly, the *DisplayFixedTokens* relation, defined by relation `DisplayFixedTokens (String, String)`, holds fixed spelling terminal symbols that are to be displayed. Fixed spelling terminal symbols that are required for presentation include brackets ('(' and ')'), binary operators ('+', '=', and ':=') and atomic statement keywords ('read', 'write' and 'call'). It should be noted that keywords that correspond to control-flow effecting statements, such as 'if' and 'while', are not collected as the structure

supplied by the control-flow graph notation necessitates the removal of such concrete syntax. Table 10 presents the relevant contents of the *DisplayFixedTokens* relation for the PL0 program of Figure 1.

String (language)	String (lexeme)
PL0	read
PL0	:=
PL0	<
PL0	+
PL0	write

Table 10: *DisplayFixedTokens* relation for the example program.

5 Analysis tool

The Analysis tool works cooperatively with the Graph Builder by using the control-flow, language semantics and presentation relations populated by the language-specific tool. By using a language-independent approach, the Analysis tool performs data-flow analysis, constructs call graphs and presents the results of both control-flow and data-flow analysis.

These activities are undertaken via a four-pass approach. In the first pass (Section 5.1), the control-flow graph is annotated with the variables modified and used by each node. Following this, the graph is further annotated with the execution signatures for each node (Section 5.2). Upon the completion of these control-flow graph annotations, the third pass performs the analysis which links the modifications and uses (Section 5.3). The final pass is for the presentation of results. The results of the flow analyses are exposed both textually and through the creation of appropriate relations to be fed back to the generic language-based editor for display using its relational navigation capabilities (Jarrott & MacDonald 2003).

The calculation of the modification and use, data-flow analysis technique is by far the most important and complex of the generic Analysis tool's tasks. These calculations form the basis of the first three passes of the Analysis tool's operation.

5.1 Pass 1: Modification-Use annotation

The first stage is where the control-flow graph is annotated to reflect the variables that are used or modified by each particular node. During this pass, the Analysis tool relies upon the information stored within the language semantics relations: *ModTokens*, *UseTokens*, *BothTokens* and *IdentTokens*. The variables modified/used for each node of the control-flow graph are determined via a state-based tree traversal algorithm. Using the non-terminal symbols defined within the language semantics relations, the tree traversal maintains a record of whether the construct is considered a modifying and/or using statement and with the assistance of the information stored in the *IdentTokens* relation (which holds the lexemes that represent identifiers in the EDL of this language) classifies all variables for each node.

5.2 Pass 2: Execution signature annotation

As the Analysis tool performs the static modification-use, data-flow analysis in a *safe* (?? ref ??) manner, it will identify a superset of nodes that potentially last modified the value of a variable. In order to determine whether a node of a control-flow graph definitely or only potentially modified the value of a variable it is required that the *execution signatures* of

each node be derived. In this research, the *execution signature* of a node is the minimum and maximum number of times a node is executed in a particular path of the control-flow graph. Since, a control-flow graph presents all paths the execution of a program can take, necessarily some choices must exist. In this context, nodes of the graph can either be marked as *compulsory* (1) meaning that it is executed in every path of the control-flow graph, or *optional* (0) meaning that it appears in a single path at a point in the program where multiple paths exist. The classifications of compulsory and optional represent the *minimum execution signature* of a control-flow graph node. Similarly, a node has a *maximum execution signature*. The maximum execution signature represents whether a node can be executed just once (*singularly*, 1) or multiple times (*multiply*, M). Together the minimum and maximum constructs are given the notation ‘minimum..maximum’, e.g., 1..1 and 0..M.

While the importance of the minimum execution signature is apparent from the safety constraint placed upon the analysis, the need for the maximum signature is less obvious. Where an iterative construct modifies the value of a variable, it is potentially possible that a successor of a node in the control-flow graph should actually be included in the set of nodes that potentially modified a variable. Figure 11 presents a code segment possessing a repetition where a node in the graph is dependent on a successor node inside the loop. Therefore, to accurately determine the full set of nodes (needed for a safe analysis), both the maximum and minimum characteristics need to be used.

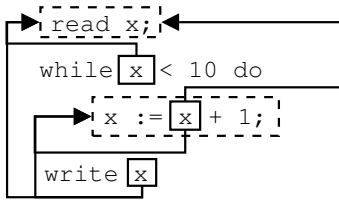


Figure 11: Example modification-use relationship for iteration

While the addition of the safety aspect to the static flow analysis also distinguishes the algorithms of this research from standard algorithms, it is the development of generic mechanisms for the calculation of execution signatures that represents the biggest change. In order to develop execution signatures for use in generic data-flow analysis, it was necessary to return to first principles and study the structures of control-flow graphs. During this study, ways of determining the execution signatures of individual nodes were developed. Basic constructs of imperative languages were classified into atomic, conditional, pre-tested repetition and post-tested repetition. The general execution characteristics of these classes were then determined (Figure 11).

Structure	Lower limit	Upper limit
Atomic statement (assignment)	1	1
Conditional (if-else)	0	1
Pre-tested repetition (while-do)	0	M
Post-tested repetition (repeat-until)	1	M

Table 11: General control structure execution limits

The basic algorithm used to calculate the execution signature of a particular node is to determine

its base signature, which defaults to 1..1 (optional, singularly executed), and then considering each of its enclosing constructs in turn (e.g., a *while-do* loop), take the minimum of its own minimum execution signature and that of the surrounding construct, and the maximum of its own maximum and that of the outer construct. One departure from the above algorithm is for the treatment of the condition in pre-tested repetition and conditional statements. These nodes do not in fact receive the minimum execution characteristic of the conditional or pre-tested iterative constructs, as in a sense they are not enclosed by those constructs, they are part of those constructs. For example, using the code presented in Figure 11, the initial signature of **W3** is 1..1, and that of the enclosing *while-do* construct is 0..M. From these figures, the actual execution signature of **W3** becomes 0..M. Figure 12 shows the control-flow graph for the code fragment from Figure 11 annotated with the execution signatures for each of the nodes.

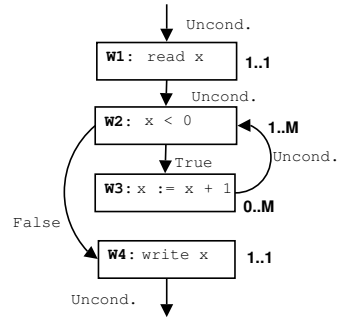


Figure 12: Graph with execution signature annotations

5.3 Pass 3: Analysis and linking

The final stage of the data-flow analysis conducted by the Analysis tool is the pass that links the uses of variables to the modifications using the graph annotations developed by the first two passes. Using recursion, this pass searches from each given node up the control-flow graph linking together (creating a UQ^* relation tuple for) a use with the set of modifications that potentially affected the current value. It must be noted that since this relationship is transitive, and the UQ^* language-based editor’s relational navigation facility nicely handles transitive relations, each use is only linked with the potential modification(s) that immediately precedes it. For example, given the code

```
C1: read b;
C2: b := b + 1;
C3: a := b
```

the use of **b** in **C3** will only be linked with **C2** and not **C1**. Using transitivity it is possible to discover that the value of **b** in **C3** is dependent on **C1** via navigating through the relational links. Figure 13 presents this diagrammatically with the modification-use relationships for these nodes.

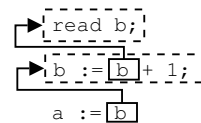


Figure 13: Chained modification-use relationships

6 Summary

Generic integration of program analysis tools into a software development environment benefits both the users and developers of such environments. The user has access to powerful tools that aid program understanding and analysis which can be invoked within the users normal working environment and minimises context switching. The user is not concerned with the generic nature of the implementation, but rather that all the languages of interest are supported in a consistent manner. The developer of software development environments is concerned with genericity as it enables the addition of said consistent support while minimising the per language workload.

In this paper it was described how a subset of flow-analysis techniques (control- and data-flow) were added to a software development environment with the goal of improving program understanding. The key contribution was not the addition of the tool support itself, but that this tool support could be added in a predominately generic manner. The design of the tools separated the functionality into language-specific and language-independent behaviour. The language-specific behaviour was codified in a Prolog-like language from which a graph building tool is automatically generated. The language-independent tool was codified in a hand-built generic analysis tool. This analysis tool required the development of two new techniques for flow analysis as traditional compiler-based techniques were found to be either overly complex, tightly coupled to other techniques or simply not suitable for program understanding due to a focus on machine code representations. Variable and statement level relationship were produced and presented to the user within the software development environment.

7 Acknowledgements

The authors would like to acknowledge the input of Phil Cook for his development of the UQ \star attribute grammar language and automatic tool generation facility used in this research.

References

- Aho, A., Sethi, R. & Ullman, J. (1986), *Compilers: Principles, Techniques and Tools*, Addison-Wesley.
- Cook, P. & Welsh, J. (2001), 'Incremental parsing in language-based editors: user needs and how to meet them.', *Software - Practice and Experience* **31**(15), 1461–1468.
- Cook, P., Welsh, J. & Hayes, I. (2002), Incremental context-sensitive evaluation in context, Technical Report 02–11, Software Verification Research Centre (SVRC), School of Information Technology, The University of Queensland, Brisbane, Australia.
- Gries, D. (1987), *The Science of Programming*, Springer-Verlag New York, Inc.
- Grundon, S., Hayes, I. & Fidge, C. (1997), Timing constraint analysis, Technical Report 97–42, Software Verification Research Centre (SVRC), School of Information Technology, The University of Queensland, Brisbane, Australia.
- Hecht, M. S. (1977), *Flow Analysis of Computer Programs*, The Computer Science Library: Programming Language Series, Elsevier North-Holland.
- Howden, W. (1982), 'Contemporary software development environments', *Communications of the ACM* **25**(3), 318–329.
- Jarrott, D. & MacDonald, A. (2003), Developing relational navigation to effectively understand software, in 'Proceedings of the Tenth Asia-Pacific Software Engineering Conference (APSEC'03)', IEEE Computer Society, pp. 144–153.
- Jones, T. & Welsh, J. (1997), 'Requirements for a generic, language-based diagram editor', *Australian Computer Science Communications* **19**(1), 316–325.
- Louden, K. (1997), *Compiler Construction: Principles and Practice*, PWS Publishing Company.
- MacDonald, A. & Welsh, J. (1999), Persistence in the UQ \star environment, Technical Report 99–43, Software Verification and Research Centre.
- Microsystems, S. (n.d.), 'Java software development platform (J2SE and J2EE)'. <http://java.sun.com/>.
- Muchnick, S. (1997), *Advanced compiler design and implementation*, Morgan Kaufmann Publishers.
- Reiss, S. (1990), 'Connecting tools using message passing in the field environment', *IEEE Software* **7**(4), 57–66.
- Rugaber, S. (1995), 'Program comprehension', *Encyclopedia of Computer Science and Technology* **35**(20), 341–368.
- Tip, F. (1995), 'A survey of program slicing techniques', *Journal of Programming Languages* **3**, 121–189.
- Toleman, M., Carrington, D., Cook, P., Coyle, A., Jones, T., MacDonald, A. & Welsh, J. (2001), Generic description of a software document environment, in R. H. Sprague, ed., 'Proceedings of the 34th Annual Hawaii International Conference on System Sciences', IEEE Computer Society. Also found in SVRC Technical Report 00-22.
- Toleman, M., Carrington, D., Cook, P., MacDonald, A. & Welsh, J. (1999), Environment description language for UQ \star , Technical Report 99–45, Software Verification Research Centre (SVRC), School of Information Technology, The University of Queensland, Brisbane, Australia.
- Web, A. (1980), 'Stoneman history'. <http://www.adahome.com/History/Stoneman>, Accessed August 2003.
- Welsh, J., Broom, B. & Kiong, D. (1991), 'A design rationale for a language-based editor', *Software - Practice and Experience* **21**(9), 923–948.
- Welsh, J. & Yang, Y. (1994), 'Integration of semantic tools into document editors', *Software - Concepts and Tools* **15**, 68–81.
- Wirth, N. (1976), *Algorithms + Data Structures = Programs*, Prentice-Hall.