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@itee.uq.edu.au

Abstract

Support for program understanding in development and maintenance tasks can be facilitated by program analysis techniques. Both control-flow and data-flow analysis can support program comprehension by augmenting the program text with additional information that may either be displayed with the program or used to allow more sophisticated navigation of the program, e.g., from an identifier's use to its definition.

This paper outlines the addition of generic program analysis support to a generic, language-based, software development environment. The analysis tools are implemented in two phases: a language-dependent phase that extracts basic information from a program, such as its control flow graph; and a language-independent phase that performs a more sophisticated analysis of the basic information. This separation allows program analysis tools to be easily generated for a new language.

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 (1995)

The use of program analysis techniques, including control-flow and data-flow analysis, has been identified as supporting software 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. For example enabling user navigation of the *definition-use* relationship for a variable allows a software developer to quickly discover both type and usage information for the variable.

The Stoneman (Howden 1982, Buxton 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 industry survey, 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.

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 flow analysis tools to support program understanding tasks.

Algorithms for flow analysis are well represented in the literature (Muchnick 1997). However for our application, 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, the modification of the techniques for flow analysis was required to enable operation on source code artefacts within the context of an integrated, generic development environment.

To reduce the costs of providing program analysis tools for multiple languages, and to allow analysis of programs written in multiple languages, the analysis tools should be generic. By generic we mean, the design of the analysis tools should be as independent of the particular language as possible, and those components that are dependent on the language should be generated from a high-level description of the language. Integrated tool-sets allow common usage paradigms for software developers across various programming languages and the tools used for those languages. Providing adequate support for efficient utilisation of tools is a major goal of software development environment research (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.

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. The attribute grammar language of Cook et. al. (2002) was used to define the language-specific aspects of the analysis tools.

In the remainder of this paper, Section 2 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⋆) used in this project is given in Section 3. Section 4 discusses the design choices made in integrating analysis, particularly generic analysis, into a software development environment. The technical aspects of the two phases of the tools developed are discussed in Sections 5 and 6. The paper closes with a summary of the outcomes of this work and potential future work.

2 Program analysis

Control-flow analysis is the study of the control structures of a program. The transformation of source code to a control-flow graph is illustrated in Figure 1. In this example, a program written in PL0 (Wirth 1976) is undergoing analysis. In addition to the derivation of control-flow graphs corresponding to a program, cycles in the graph can be identified (as described in Muchnick (1997)). The control-flow graphs along with information identifying their cycles can then be used as the basis for more advanced program analysis techniques, including data-flow analysis.

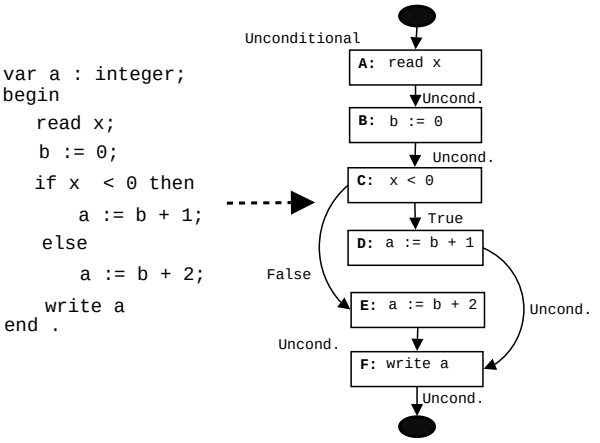

Figure 1: Example program and control-flow graph

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

3 Software development environments

Work at The University of Queensland has produced an experimental software development environment, UQ⋆ (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). Software documents (Welsh 1994) in this context consist of both abstract syntax trees and relations. UQ⋆ is a generic language-based environment in that, when provided with appropriate descriptions of the languages, it provides its users with language-specific support. Toleman et. al. (2001) defines the Environment Description Language (EDL) used to describe languages and presentations. EDL supports UQ⋆'s multi-lingual nature and provides mechanisms for defining textual and graphical languages, relations, and semantic tools.

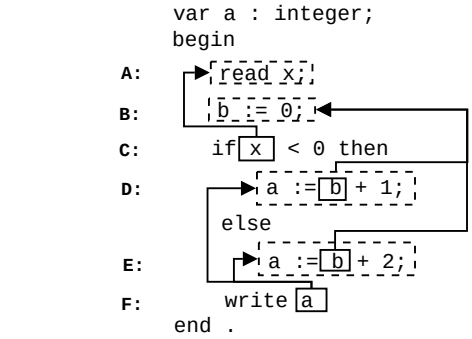

Figure 2: Example modification-use chain

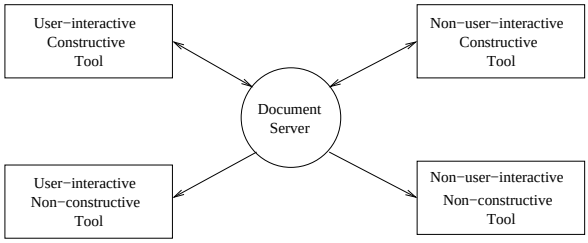

Figure 3: UQ ⋆ architecture

The UQ⋆ architecture is shown in Figure 3. A central document server manages the syntactic and relational structures and ensures their persistent storage (MacDonald & Welsh 1999).

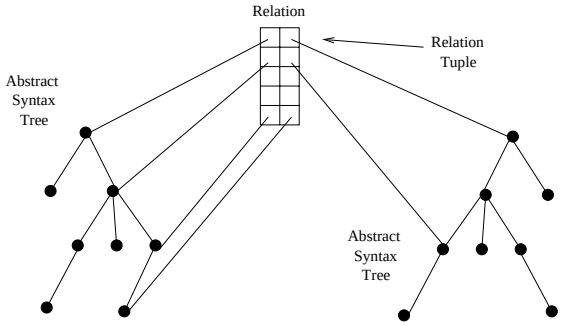

Figure 4: UQ⋆ syntactic structures

Syntactic structures are represented as abstract syntax trees. These are augmented with n-ary relations (as shown in Figure 4) to represent connections between elements in the abstract syntax tree. Relational elements can be abstract syntax trees, nodes within abstract syntax trees, relations, strings or numeric data. In general, UQ⋆ relations are N-ary and bi-directional; hence the underlying model of UQ⋆ relations is more akin to that used in relational databases.

The document server makes these structures accessible to a collection of tools (Welsh & Yang 1994). These tools can be categorised depending on whether or not they *interact* with the user, and whether or not they *construct* syntactic and relational structures. A language-based editor is an example of an interactive and constructive tool (Cook & Welsh 2001). Interactive and non-constructive tools are less common, but an example is a read-only preview of a document, or a non-editable view, such as a call-graph view of a program. A static semantic analyser is a back-end tool that generates relational information linking syntactic constructs of its input document with error messages and is thus considered an example of a non-interactive, constructive tool. Exporting a pretty-printed version of a document to postscript falls within the category of non-interactive and non-constructive tools.

A recent addition to the UQ⋆ programming support environment is the automatic specification of relation building tools via the use of a meta-language (Cook, Welsh & Hayes 2002) extending the existing environment description language. The meta-language consists of an attribute grammar (Knuth 1968) in which attributes may be relations. The definition of relational attributes uses a Prolog-like language syntax and recursion to implement potentially complex abstract syntax tree traversals. Tools created in this manner can both *use* existing relations and abstract syntax trees and *provide* new relations and abstract syntax trees to the system for utilisation in other components.

4 Design

In this research, a system was developed to conduct flow analysis in a manner as independent of the language being analysed as possible. For simplicity, this paper focuses on the flow analyses as they apply to the language PL0 (Wirth 1976). Two UQ⋆ tools, the Graph Builder (Section 5) and the Analysis tool (Section 6) were developed. Using the tool classifications of Section 3, both of these tools can be categorised as non-interactive and constructive. The use of *global* (stored in the UQ⋆ document server) relations allows collaboration between the two tools, in addition to making the results of the flow analyses available for use by other UQ⋆ tools.

The first of the tools is the language-specific Graph Builder. The Graph Builder is written using UQ⋆'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 presentation of results) and abstract syntax (for language semantics such as which non-terminals can modify and/or use the values of variables) needed for use in the second tool, the Analysis tool.

The second of the tools is the language-independent Analysis tool. The Analysis tool uses the relations populated by the Graph Builder to create call-graphs, conduct data-flow analysis (to determine the modification-use relationships) and present the results of both forms of flow analysis to the user. The effort required to set up flow analysis for additional languages has been minimised by making as much of the system as language-independent as possible. Additionally, the desire for language-independence is directly responsible for the decision not to implement the data-flow analysis in the EDL attribute grammar.

The tools were developed for the UQ⋆ generic, integrated development environment. From an integrated development environment perspective, it is important that the tools place minimal requirements on the system that they are to work with. This includes conforming to existing usage paradigms for UQ⋆. This requirement dictated that the tools would be generic and incremental where possible. It was the desire to work within the existing UQ⋆ environment that motivated the creation of the two phase approach. By making the Analysis tool language-independent, 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 works with the PL0 (Wirth 1976) language variant defined for use in UQ⋆. Unfortunately the grammar used for PL0 in UQ⋆ is not optimal for conducting data-flow analysis, especially using a generic approach. Rather than require changes to the grammar, which could potentially impact on existing tools developed for the PL0 language, the direction chosen was to develop methods that were flexible enough to work with existing language designs.

5 Graph Builder

It is the Graph Builder that is responsible for the creation and population of the relations representing a program's control-flow graph. The Graph Builder tool was defined using an attribute grammar language (Cook et al. 2002) and compiled into an executable binary (via C++ code) using the EDL compiler. 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 6). The Graph Builder interacts within the UQ⋆ system in a constructive, non-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 created by the Graph Builder are used to model the control-flow graph of a given program as well as extract other information required for use in the language-independent Analysis tool.

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

5.1 Control-flow Graph

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 represents the UQ⋆ document server relations that result from the calculations of the Graph Builder tool. The calculation of a program's control-flow graph is undertaken using two levels of UQ⋆ relations. The first level are the synthesized attribute grammar relations which, via a bottom-up process,

construct the flow graph of the program from its abstract syntax tree. The attribute grammar relations are internal to the Graph Builder tool, and are used to populate the second level of relations corresponding to the *Graph* relations. The relations at both levels have been kept structurally equivalent to simplify processing within the tool, however they are distinguished via their names. For example, the *GraphTrans* relation, described next, is the UQ⋆ document server relation responsible for maintaining the flow edges of a control-flow graph. The *GraphTrans* relation is structurally equivalent to the *trans* relation which is used during the calculation process and is internal to the Graph Builder tool.

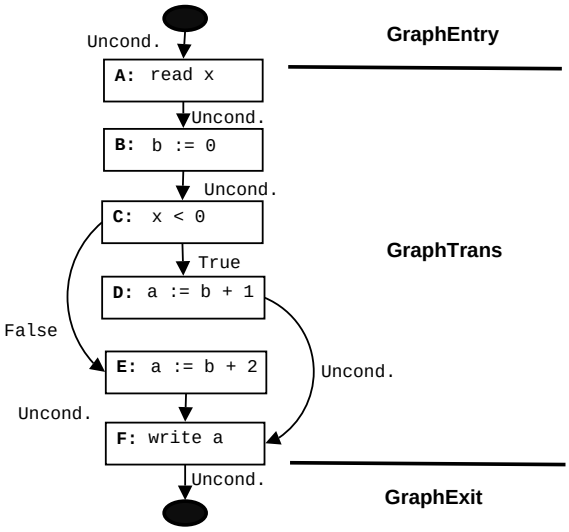

Figure 5: Control-flow graph relations

The *GraphTrans* relation represents the internal edges of the control-flow graph. It is the fundamental component of the Graph Builder's final output. Each 3-tuple in the *GraphTrans* relates a node to its successor with an labelled edge. In EDL, *GraphTrans* is declared as:

```
Relation GraphTrans (Node, Label, Node).
```

Within the current configuration there are three possible edge labels: *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 defined to allow extensions for unrestricted branching including multiple branching `case` statements, 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 invocation of the Graph Builder are displayed in Table 1.

Node (from)	Label	Node (to)
A	*Unconditional*	B
B	*Unconditional*	C
C	*True*	D
C	*False*	E
D	*Unconditional*	F
E	*Unconditional*	F

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).
```

Table 2 shows the contents of the *GraphExit* relation for the program of Figure 1.

Node	Label
F	*Unconditional*

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

A control-flow graph may possess multiple exit points, or a conditional exit point. A conditional exit point (an exit on *True* or *False*) may occur where the last statement of a program is an iterative construct (i.e., a `while` loop). For example, a program which ends with a `while` loop will have an exit on *False*. Figure 6 shows an example of a program with a conditional exit; the contents of the *GraphExit* relation for this program are shown in Table 3.

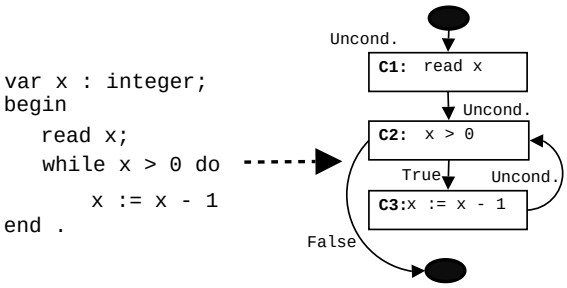

Figure 6: Program with a conditional exit

Node	Label
C2	*False*

Table 3: *GraphExit* relation for the program with a conditional exit.

Multiple exits may occur where a conditional statement is the last statement of a program/procedure, and in languages such as Java, where a method can have multiple `return` statements. Figure 7 provides an example of program with two exit points. Table 4 shows the contents of the *GraphExit* relation for this example.

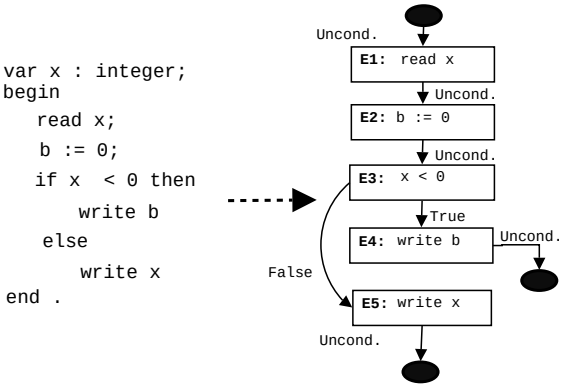

Figure 7: Program with multiple exits

The *GraphEntry* relation is defined as

```
relation GraphEntry (Label, Node)
```

It 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

Node	Label
E4	*Unconditional*
E5	*Unconditional*

Table 4: *GraphExit* relation for the program with multiple exits.

a labelled edge, however unlike the other control-flow relations, each flow graph possesses only one entry. The *GraphEntry* label may seem redundant, but it is necessary to support multiple-branching statements (e.g., `case` statements). Table 5 shows the contents of the *GraphEntry* relation for the example program of Figure 1.

Label	Node
Unconditional	A

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

5.1.1 Procedures

The Graph Builder maintains two 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*). Inter-procedural analyses are complicated by analysing across abstract syntax trees, and by procedure parameters, which introduce the possibility of variable aliasing, and variable scope (Tip 1995). Static analysis is further complicated by recursion.

These issues provided the motivation for the separation into two sets of relations. Furthermore, it was decided that by making the structure of the procedure and main method relations similar, a single traversal algorithm could be used in the Analysis tool (Section 6).

In addition to the *GraphProcEntry*, *GraphProcExit* and *GraphProcTrans* relations, the Analysis tool requires a relation mapping the name of each procedure to the nodes representing the entry and exit nodes 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:

```
relation GraphProcMap (String, Node, Node).
```

Its parameters correspond to the name of the procedure, and its entry and exit nodes. Where a procedure possesses more than one exit node, an tuple is included for each distinct exit. It is a constraint of the system that the entry and exit nodes within *GraphProcMap* must exist within the *GraphProcEntry* and *GraphProcExit* relations.

5.2 Relation Population

The Graph Builder tool is generated from the compilation of the `GraphBuilder.edl` file. This file defines an attribute grammar, with a Prolog-like syntax for populating relation tuples. This grammar is used to calculate the control-flow graph of a program, and identify semantic information for data-flow analysis which is specific to the language the program is written in. As the EDL file provides the language-specific information, it is the portion of the flow analysis system that requires definition for each additional language. The EDL file provides *phylum*s which define the inherited and synthesised attributes for each non-terminal, and the *rules* that specify how the attributes are evaluated. The semantics of the EDL require that for every non-terminal defined in a language, a *phylum* must be declared. Additionally, for every alternative 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. Figure 8 provides the example of the `Statement` non-terminal from the PL0 language defined in Toleman et al. (2001).

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

(a) Language EDL

```
phylum Statement
{
attributes:
  relation entry (Label, Node) .
  relation exit  (Node, Label) .
  relation trans (Node, Label, Node) .
defaults: ...
}

rule Statement = ReadStatement
{    ...   }

rule Statement = IfStatement
{    ...   }

...
```

(b) Attribute grammar EDL

Figure 8: Language, phylum and rule definition for `Statement`

The attributes of an AST node used to calculate the control-flow graph are the relations *entry*, *trans* and *exit*. These relations have the same type as the global *GraphEntry*, *GraphTrans* and *GraphExit* relations.

Two classes of constructs can be identified: *primitive* (or *atomic*) such as an assignment statement, and *structured* such as statement lists, branching and iterative constructs including `if` and `while` statements. It is the structured statements that define the control-flow graph of a program.

Primitive constructs are characterised by having no control-flow transitions (an empty *trans* relation). For example, a condition (e.g., `x < 0`) is an atomic structure.

```
rule Condition = Exp
{
    this->entry(UNCOND, this).
    this->exit(this, UNCOND).
    this->trans = empty.
    /* explained in Section 5.3 */
    Relations::UseTokens("PL0", "Condition",
                         "Ident") .
}
```

Figure 9: Specification for Condition primitive construct.

Figure 9 is the EDL rule specification for a `Condition`. From the *entry* and *exit* relations defined in the `Condition` rule, the rule for the `IfStatement` in Figure 10 can assert that the *entry* relation attribute of the `IfStatement` node ("`this`") includes a tuple `entry(UNCOND,n)` whenever the *entry* relation of the `Condition` node includes such a tuple, i.e., the *entry* relation of the `IfStatement` node includes the *entry* relation of the `Condition` node, which only has a single tuple. The syntax for a clause of the form

```
forall(Node n)
    this->entry(UNCOND,n)
        :- Condition->entry(UNCOND,n)
```

is Prolog-like. However, instead of using Prolog's convention that all upper case identifiers are implicitly universally quantified and lower case identifiers are constants, the notation used is case insensitive and variables are explicitly quantified. The explicit quantification also specifies the type of the variable. Because this is the only clause defining the *entry* relation for the `IfStatement` node, this is the only tuple contained in the relation. This can be compared with the definition of *exit* relation which requires two clauses: the first specifying that the *exit* relation of the `IfStatement` node contains the *exit* relation of the `Statement` node `s1`, and the second that it contains the *exit* relation of `s2`. For the *exit* relation the labels may be either unconditional (`UNCOND`) or conditional (`TRUE` or `FALSE`) depending upon the form of the exit from the component statements `s1` and `s2`.

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

Figure 10: `if` statement rule definition

The attribute grammar used to compute the graph follows a bottom-up approach. For a structured programming language such as PL0, this means that the graph of a structured construct (a conditional or iterative construct) is built using the attributes of its lower level components. For the `IfStatement` the rule asserts that there are transitions from the condition component to the nodes that form the entry points of the component statements (`s1` and `s2`), and that the transitions are labelled *true* and *false* respectively. Finally, the rule completes the definition of *trans* by adding all nested transitions within `s1` and `s2` to the overall set of transitions.

The transitions for the sequences are built up using the entry and exits of its 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 is also given in the EDL definition. The population of these relations is demonstrated in Figure 9 with the addition of a tuple to the global *UseTokens* relation (explained in Section 5.3), signifying that a condition can use the value of a variable at any point within the construct. It is the information held in these relations that enables the approach to data-flow analysis to be generic.

5.3 Language Semantics

Another class of global Graph Builder relations are those that capture the semantics of a given language for use in the generic Analysis tool. These relations hold the 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, effectively means that flow analysis tool can be multi-lingual (i.e., it can conduct the analyses on programs from a diverse range of languages in one session).

The three relations, *ModTokens*, *UseTokens* and *BothTokens*, 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. For example, given the statement `x := y + z`, the value of `x` is modified and the values of both `y` and `z` are used.

The *ModTokens*, *UseTokens* and *BothTokens* relations (explained in turn below) are structurally equivalent, and are of type `(String, String, String)`. 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 element specifies the name of the non-terminal symbols from the language that represent the identifier that is either modified and/or used. Moreover, constructs can modify and use multiple identifiers.

The *BothTokens* relation is used for language constructs that both modify and use a variable using a single non-terminal, for example the `i++` and `i += 10` syntax from C. The *ModTokens*, *BothTokens* and *UseTokens* relations are pairwise disjoint. The tuples that appear in the *BothTokens* relation do not appear in either of the *ModTokens* and *UseTokens* relations, and while *BothTokens* could have been computed from the intersection of all modifications and all uses, for performance reasons a separate relation was maintained. Using the current relations the union of *ModTokens* and *BothTokens* will give the set of all modifications, similarly the union of *BothTokens* and *UseTokens* will give the set of all uses.

6 Analysis tool

The Analysis tool uses the control-flow and language semantics relations defined by the language-specific Graph Builder tool. The chosen representations mean that the Analysis tool is language independent. The Analysis tool performs data-flow analysis, constructs call graphs, and presents the results of both control-flow and data-flow analyses.

These activities are undertaken via a four-pass approach. In the first pass (Section 6.1), 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* (defined in Section 6.2) for each node. The third pass performs an analysis which links the modifications and

uses (Section 6.3). The final pass is for the presentation of results. The results of the flow analyses are presented 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).

6.1 Pass 1: Modification-Use annotation

In the first stage, 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*, and *BothTokens*. 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 construct, and classifies all variables for each node.

6.2 Pass 2: Execution signature annotation

We have defined a *safe* analysis that determines the full set of nodes that *may* have affected the values of a variable. In some cases it may include nodes that cannot affect the value of a variable at some later node because the path between them cannot be followed at runtime - it is a dead path. As the Analysis tool performs the static modification-use analysis in a *safe* 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. The *execution signature* of a node is the minimum and maximum number of possible 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, multiple paths between a source and target node may 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 (1) or multiple times (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 pessimistic nature of the analysis, the need for the maximum signature is less obvious. Where the value of a variable is modified within an iterative construct, nodes occurring within the iterative construct but before the modification are also dependent on the modifying statement (for their value on the next iteration), and should be included in the set of modifying statements for that node. For example, in Figure 11 the uses of `x` in **W2**, **W3** and **W4** depend on the modification of `x` in **W4**. Therefore, to accurately determine the full set of nodes (needed for a safe analysis), both the maximum and minimum characteristics need to be used.

It is the development of generic mechanisms for the calculation of execution signatures that distinguishes the algorithms of this research from standard optimising compiler algorithms. In order to develop

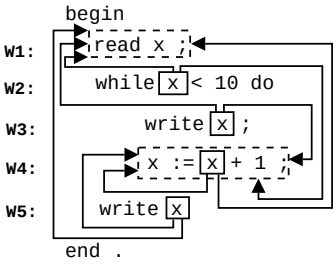

Figure 11: Example modification-use relationship for iteration

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 can be classified into atomic, conditional, pre-tested repetition and post-tested repetition. The general execution characteristics of these classes are given in Table 6.

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 6: 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` (compulsory, single execution), and then consider each of its enclosing constructs in turn (e.g., a `while` 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 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` 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.

6.3 Pass 3: Analysis and linking

The final stage of the data-flow analysis links all uses of a variable to the modifications of the variable that may have affected its value at that point. The analysis and linking pass uses recursion to search from a node up the control-flow graph (towards the graph's entry), linking together a use with its set of modifications. It must be noted that since the modification-use 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 modifications that immediately precede it. For example, given the code:

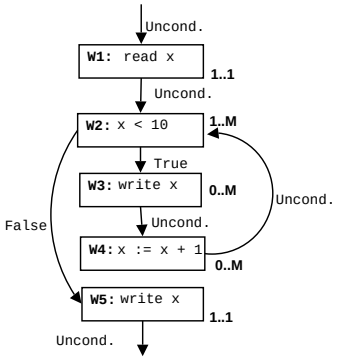

Figure 12: Graph with execution signature annotations

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.

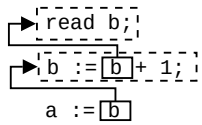

Figure 13: Chained modification-use relationships

6.4 Finalisation

Upon completion of the analysis and linking pass, the Analysis tool populates the *Mod-Use* UQ⋆ Document server relation. This relations is defined as:

```
relation Mod-Use (String, Node, Node).
```

The parameters correspond to the variable that is being used, the AST node that represents a last (or potential last) modification and the AST node at which the use is occurring. Table 7 shows the contents of the *Mod-Use* relation for the example program of Figure 1.

Variable	Mod Node	Use Node
`x`	A	C
`b`	B	D
`b`	B	E
`a`	D	F
`a`	E	F

Table 7: *Mod-Use* relation for the example program

Table 7 also gives an example of a use (of variable `a` at node F) which has two possible modifications (at nodes D and E).

7 Summary

This paper described the addition of control-flow and data-flow analysis to a generic software development environment to support improved 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 using an attribute grammar language, from which a graph building tool is automatically generated. The language-independent behaviour was codified in a generic analysis tool. The Analysis tool performs a safe data-flow analysis using the execution signatures of the control-flow graph nodes built by the Graph Builder tool. A modification of traditional data-flow analysis technique was required 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. From the output of these two tools, variable and statement level relationships were presented to the user within the software development environment.

The capabilities provided by this work will allow techniques that rely on the results of control-flow analysis, such as Timing-Constraint Analysis (Grundon, Hayes & Fidge 1998) to be implemented in UQ⋆ in future projects. This work is also a first step in developing tools to support the modernization of legacy systems such as those required as part of the Object Management Group's Architecture-Driven Modernization project (Object Managment Group ADM Taskforce 2003, Gerber, Glynn, Lawley, MacDonald & Raymond 2004).

Integration of program analysis tools into a software development environment benefits the users 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 consistent support while minimising the per language effort.

Acknowledgements

The authors would like to acknowledge the input of Phil Cook for his development of the UQ⋆ attribute grammar language and automatic tool generation facility used in this research. The authors would also like to thank Kerry Raymond for her insightful comments on earlier drafts of this paper.

References

Buxton, J. (1980), DoD requirements for Ada programming support enviroments, STONEMAN, Technical Report AD-A100 404, US Department of Defense. Available from `http://www.adahome.com/History/Stoneman`, Accessed August 2003.

Cook, P. & Welsh, J. (2001), 'Incremental parsing in language-based editors: User needs and how to meet them.', *Software - Pratice 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.

Gerber, A., Glynn, E., Lawley, M., MacDonald, A. & Raymond, K. (2004), Knowledge Discovery MetaModel - Initial Submission, OMG Submission admtf/04-04-01, Object Management Group.

Grundon, S., Hayes, I. & Fidge, C. (1998), Timing constraint analysis, *in* C. McDonald, ed., 'Proceedings 21st Australasian Computer Science Conference', pp. 575–586.

Hecht, M. S. (1977), *Flow Analysis of Computer Programs*, The Computer Science Library: Programming Language Series, Elsevier North-Holland.

Howden, W. (1982), 'Comtemporary 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, *in* M. Patel, ed., 'Proceedings of 20th Australian Computer Science Conference', Vol. 19, pp. 316–325.

Knuth, D. (1968), 'Semantics of context-free languages', *Math. Systems Theory* **2**(2), 127–145.

MacDonald, A. & Welsh, J. (1999), Persistence in the UQ$\star$ environment, Technical Report 99–43, Software Verification and Research Centre, School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane, Australia.

Muchnick, S. (1997), *Advanced compiler design and implementation*, Morgan Kaufmann Publishers.

Object Managment Group ADM Taskforce (2003), Request for Proposal - Architecure-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM), OMG RFP lt/03-11-04, Object Management Group.

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.

Welsh, J. (1994), Software is history!, *in* A. W. Roscoe, ed., 'A Classical Mind: Essays in honour of C.A.R. Hoare', Prentice Hall, chapter 24, pp. 419–430.

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.