A Language for the Description of Program Instrumentation and Automatic Generation of Instrumenters

Adenilso S. Simão
José C. Maldonado
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo
São Carlos, São Paulo, Brazil
{adenilso, jcmaldon}@icmc.usp.br

Auri M. R. Vincenzi
UNIVEM — Centro Universitário Eurípides de Marília
Fundação de Ensino Eurípides Soares da Rocha
Marília, São Paulo, Brazil
auri@fundanet.br

Antônio C. L. Santana
UniLins — Centro Universitário de Lins
Fundação Paulista de Tecnologia e Educação
Lins, São Paulo, Brazil
santana@fpte.br

Abstract

Instrumentation is a technique frequently used in software engineering for several different purposes, e.g. program and/or specification execution trace, testing criteria coverage analysis, and reverse engineering. Instrumenting a software product can be divided into two main tasks: (i) deriving the software product structure and (ii) inserting statements for collecting runtime/simulation information.

Most instrumentation approaches are specific to a given domain or language. Thus, it is very difficult to reuse the effort expended in developing an instrumenter, even if the target languages are quite similar. To tackle this problem, in this paper, we propose an instrumentation-oriented meta-language, named IDeL, designed to support the description of both main tasks of instrumentation process, namely: (i) the product structure derivation and (ii) the insertion of the instrumentation statements. In order to apply IDeL to a specific language L, it should be instantiated with a context-free grammar of L. To promote IDeL's practical use, we also developed a supporting tool, named `idelgen`, that can be thought of as an application generator, based on the transformational programming paradigm and tailored to the instrumentation process. We illustrate the main concepts of our proposal with examples describing the instrumentation required in some traditional data flow testing criteria for C language.

Keywords: Software Engineering, Programming Languages, Program Instrumentation.

1 Introduction

Many activities in the software development process involve the analysis of program code, for instance in reverse engineering, program visualization, debugging and testing. In general, the analysis of a program includes scanning it, retrieving useful information and producing an abstract view of its main elements. One important kind of abstract view we can derive from a program is its control and data flow information [14]. The program (or a function thereof) is represented as a graph, usually refereed to as "program graph", whose nodes and edges are, respectively, the program's statement blocks and the possible control transference between blocks. The relevant information about manipulation of variables is attached to nodes or edges of the program graph.

In addition to deriving the program graph, it is often also necessary to inspect the actual program execution. For example, in order to assess conformance to some testing coverage criteria, one may want to know which statements were executed and in which order. Analogously, considering the program graph, one may want to know which paths are traversed. With this purpose, special log statements are inserted into the program, producing the so-called *instrumented program*. These log statements register the execution information in a trace file, without changing the program semantics. Taken together, program graph derivation and log statement insertion are often referred to as *program instrumentation*. Being a lengthy and error-prone process, program instrumentation is usually performed in an automatic way by a so-called *instrumenter*.

An instrumenter analyzes the program according to the grammar of its language, recognizes the relevant structures of the program, and derives both the program graph and the instrumented program based upon the semantics of the language. Thus, some techniques and tools employed in the construction of compilers are usually useful for the development of instrumenters. For example, the analysis of the program structure can be done with compiling tools like `yacc` and `lex` [12]. The drawback of this approach is the difficulty in reusing the effort expended in a past development of an instrumenter in the development of other instrumenters. Even in the case of similar languages, each new instrumenter will demand a complete development cycle.

Another approach for the development of instrumenters is the use of transformational systems, such as TXL [6] and Draco [13]. However, those systems are not tailored to instrumentation and, the development of an instrumenter demands a substantial effort, requiring the definition of suitable data structures and statements. Moreover, this approach has the same drawback of the previous one, that is, the difficulty in reusing.

In this paper, we propose a system to support the instrumentation process. Our approach can be thought of as an *instrumenter generator*, based on the transformational paradigm and graph theory and restricted to the program instrumentation domain. The system consists of (i) a language, named IDeL (**I**nstrumentation **De**scription **L**anguage), and (ii) a "compiler" (named `idelgen`, standing for IDeL **Gen**erator) that will generate an instrumenter based on the IDeL description. The IDeL language embodies constructions suitable for describing the features of the intended instrumenter. It aims at achieving a good trade-off between generality and complexity. It hides the complexity of traditional transformational languages. In order to describe an instrumenter in the IDeL language, one has to write down only the features that are specific to the application in mind. `idelgen` will automatically provide the data structures (and the respective statements) that are common to any instrumenter.

IDeL is part of a larger effort we are endeavoring to specify, design and implement a generic, multi-language tool for supporting Mutation Testing [7] and control/data flow testing criteria [11, 14]. Our work involves the definition of a general framework, as well as mechanisms to instantiate it for a particular purpose. Thus, our primary motivation in developing IDeL is to provide the basis of these mechanisms with respect to (w.r.t.) testing criteria, enabling us to derive program graphs [14] and instrumented programs in order to trace their executions. So far, we have been able to use IDeL in some case studies, such as C, C++ and Java program instrumentation for testing coverage analysis and program visualization.

This paper is organized as follows. In Section 2 we present work related to our approach. In Section 3 we introduce background concepts of program graphs and grammar theories. In Section 4 we present the main features of IDeL. In Section 5 we describe `idelgen`, a supporting system for generating instrumenters based on an IDeL description. Finally, in Section 6 we make some concluding remarks and point future work.

2 Related Work

Several software engineering activities use instrumentation as a mechanism to support program analyses. Moreover, instrumentation is often used to trace program execution. In this section, we discuss some work related to instrumentation.

Kotik and Markosian [10] describe a piece of work carried out with a transformational system — called Refine — to generate test cases. In their approach, the authors provide a means to derive boundary values appropriate to each suitable control structure of the program. The program is, then, changed in order to incorporate the drivers for testing these boundary values. Observe that, although Kotik and Markosian's approach indeed changes the program with a purpose different from instrumentation, it is related to our work in the sense that a transformational based system is employed to handle programs and support the testing activity.

The model behind data flow criteria definitions [11, 14] is the *def-use graph*, which is a program graph enriched with information about the definitions (assignments) and uses of variables attached to its nodes and edges. The def-use graph of a particular program is, then, analyzed in order to derive the testing requirements. The program is instrumented with statements to register the execution flow, which is used to verify whether the testing requirements were exercised by a given test case set. In [11], Maldonado proposes data flow testing criteria that are based on *potential* uses of variables. These testing criteria also employ program instrumentation and use a slightly different version of the def-use graph. These criteria were implemented in the tool Poke-tool [5].

In [4], Bueno and Jino use program instrumentation in an approach for employing genetic algorithms to generate test cases. In that work, besides information about the number of nodes, definitions and uses of variables, the fitness function of a genetic algorithm requires other kinds of information, e.g. the value of expressions related to a predicative node after the execution of a test case. The way the instrumentation is carried out allows to collect the required information to automate the test case generation.

There are cases in which either program alteration or graph derivation is required, but not both. For example, program instrumentation made for performance analysis only includes that will collect useful information about the dynamic aspects of a particular execution, allowing the

identification of performance problems, such as bottlenecks [1]. For this case in particular, the program graph is not necessary. On the other hand, the abstract view provided by the program graph is often more appropriate in tasks that require an overall insight of the program structure, e.g. reverse engineering. For this case, the program does not need to be altered.

Program instrumentation is, therefore, related to several software engineering activities, in particular, to activities of Validation, Verification and Testing. In general, different types of instrumentation are required according to the objective pursued. In this way, an instrumentation tool should be flexible and, at the same time, suitably tailorable, in order to demand a minimum instantiation effort.

3 Basic Concepts

In this section, we present background concepts related to program graphs and grammars, needed for the understanding of the remaining of this paper.

3.1 Program def-use Graphs

The structure of a program can be represented in a so-called *program graph*. The topology of the graph reflects the possible flows of control that can occur in an actual execution of the program. The graph nodes represent the relevant components of the program. Usually, those components are the executable statements, although other kinds of components may be represented as well, e.g. declaration and function definitions. The graph edges represent the relationships between the components (the graph nodes). An edge from node a to node b means that b may be executed after a in some execution of the program.

Consider a set S of statements that share the property that, in any normal execution of the program, when the first of them is executed, all the others will also be executed in sequence. In other words, the control flow can only branch to the first statement of S, or from the last one. Such a set is called a *statement block*. Therefore, a program can be completely represented by a graph whose nodes are the distinct statement blocks and whose edges are the flows to the beginning and from the end of statement blocks.

A def-use graph is an extension of the program graph and includes information about the variables that are defined or used. It is often used as the underlying representation for most of the data flow testing criteria (e.g. [11, 14]). There are three kinds of information:

variable definitions: A variable definition occurs in the points of the program where a value is assigned to a variable.

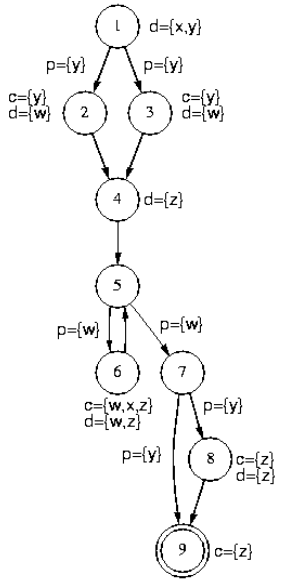
variable uses: A variable use occurs in the points of the program where a value assigned to a variable is used in some expression. The variable uses are partitioned in:

- **predicative uses:** A predicative use (p-use) is a variable use that occurs in an expression which controls the execution flow.
- **computational uses:** A computational use (c-use) is a variable use that is not a predicative use, i.e. a variable use that does not occur in a control expression.

In order to illustrate the concepts presented so far, Figure 1 presents a simple C program and its corresponding def-use graph. Actually it presents a single function that computes $z = x^y$. We omit the operational details (e.g. the main function required in a C program), since they are not relevant in this example. The block that corresponds to a statement is written in comments. For example, we can observe that node 6 is a block statement composed of two statements. The graph also includes information about the definitions, p-uses and c-uses of variables, abbreviated to `d`, `p` and `c`, respectively. For example, a p-use of the variable `w` in the control expression of the *while* statement (node 5) is represented by the two labels near to the edges $(5, 6)$ and $(5, 7)$, which are the possible control flow branches in this point.

```
float power (float x, int y) {
   float z;
   int   w;
   if (y > 0) {          /* 1 */
      w = y;             /* 2 */
   } else {
      w = -y;            /* 3 */
   }
   z = 1.0;              /* 4 */
   while (w != 0) {      /* 5 */
      z *= x;            /* 6 */
      w--;               /* 6 */
   }
   if (y < 0) {          /* 7 */
      z = 1 / z;         /* 8 */
   }
   return z;             /* 9 */
}
```

(a)



(b)

Figure 1: (a) A simple function that calculates $z = x^y$ and (b) its corresponding def-use program graph, extracted from [14].

3.2 Grammars and Syntax Trees

Syntax grammars are finite devices that are often used to describe infinite languages. Given a grammar G, we let $L(G)$ be the set of all sentences that can be generated by the productions in G. Most, if not all, programming languages are characterized by a grammar. Indeed, the grammar is usually part of the sound definition of the language. Grammars can be classified based on the kind of productions they possess. An important class is the *context-free grammars*. They are simple but expressive enough to catch most constructions that are usually found in programming languages. Moreover, the algorithms to recognize them are computationally tractable. Context-free grammars are usually described in BNF [19]. We will refer to them as BNF grammars, as a shortcut for context-free grammar described in BNF. A BNF grammar G is formed by a four-tuple $G = (N, T, S, R)$, where N is the set of non-terminal symbols, T is the set of terminal symbols, $S \in N$ is a non-terminal symbol referred to as the initial symbol, and $R \subseteq N \times (N \cup T)^*$ is the

set of production rules. A production rule of the form (n, α) states that the non-terminal symbol n (the *left-hand* symbol) can be replaced by the sequence α (the *right-hand* symbol sequence) of terminal and non-terminal symbols without "inflicting" the grammar.

As an example of a BNF grammar, Figure 2 presents a grammar that defines a simple language. This language includes *if* and *while* statements. The non-terminal set of this grammar is $N = \{\langle S\rangle, \langle SL\rangle, \langle W\rangle, \langle IF\rangle, \langle E\rangle, \langle ID\rangle, \langle C\rangle\}$ and the terminal set is $T = \{$'`break`', '`while`', '`if`', '`else`', '`(`', '`{`', '`}`', '`)`', '`;`', '`+`', '`-`', '`>=`', '`=`', '`,`'$\}$. The initial symbol $S = \langle S\rangle$. For sake of simplicity, we do not specify the productions for $\langle ID\rangle$ and $\langle C\rangle$. We assume them to be, respectively, any identifier and integer symbols valid in C language.

$\langle S\rangle$ ::= $\langle W\rangle$
$\langle S\rangle$ ::= $\langle IF\rangle$
$\langle S\rangle$ ::= '`break`' '`;`'
$\langle S\rangle$ ::= $\langle ID\rangle$ '`=`' $\langle E\rangle$ '`;`'
$\langle S\rangle$ ::= '`{`' $\langle SL\rangle$ '`}`'
$\langle SL\rangle$ ::= $\langle S\rangle$
$\langle SL\rangle$ ::= $\langle SL\rangle$ $\langle S\rangle$
$\langle W\rangle$ ::= '`while`' '`(`' $\langle E\rangle$ '`)`' $\langle S\rangle$
$\langle IF\rangle$ ::= '`if`' '`(`' $\langle E\rangle$ '`)`' $\langle S\rangle$
$\langle IF\rangle$ ::= '`if`' '`(`' $\langle E\rangle$ '`)`' $\langle S\rangle$ '`else`' $\langle S\rangle$
$\langle E\rangle$ ::= $\langle ID\rangle$ '`>=`' $\langle C\rangle$
$\langle E\rangle$ ::= $\langle ID\rangle$ '`+`' $\langle ID\rangle$
$\langle E\rangle$ ::= $\langle ID\rangle$ '`-`' $\langle ID\rangle$
$\langle E\rangle$ ::= $\langle ID\rangle$ '`(`' $\langle E\rangle$ '`)`'
$\langle E\rangle$ ::= $\langle E\rangle$ '`,`' $\langle E\rangle$
$\langle ID\rangle$ ::= some identifier
$\langle C\rangle$ ::= some integer

Figure 2: A BNF grammar of a sub-language of C.

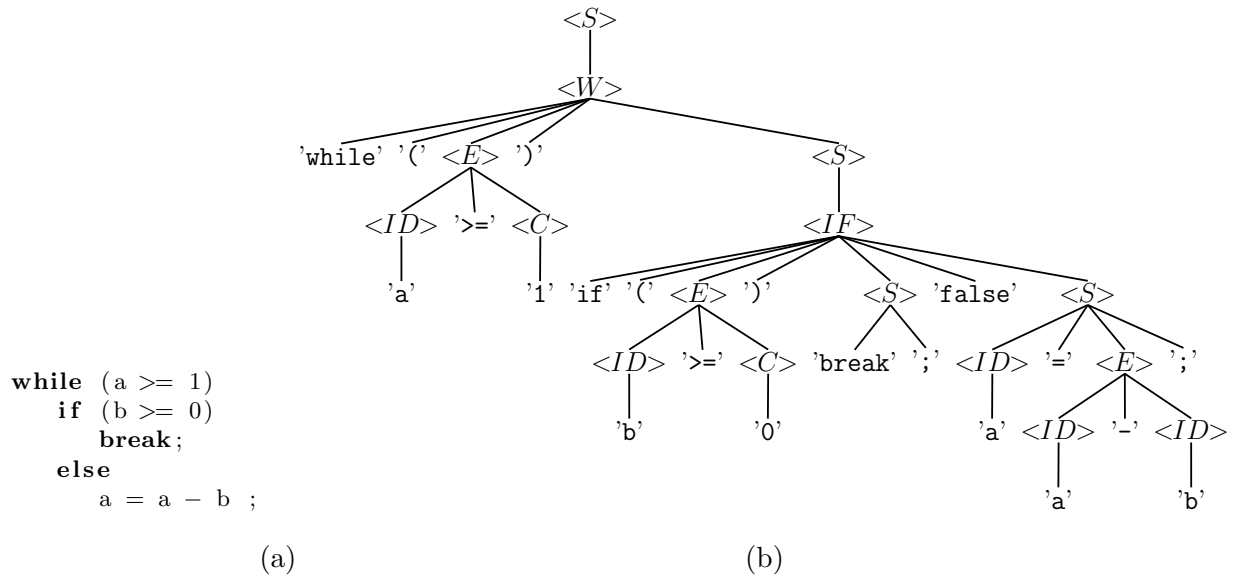
From a sequence $\gamma\langle n\rangle\delta$, we can derive another sequence of the form $\gamma\alpha\delta$, for any production (n, α). This is represented by

$$\gamma\langle n\rangle\delta \Rightarrow \gamma\alpha\delta$$

The language $L(G)$ defined by G is the set of all sequences of *terminal* symbols that can be derived from the *initial symbol* S with the productions in R, i.e., $\varphi \in L(G)$ if and only if $\varphi \in T^*$ and $S \Rightarrow \ldots \Rightarrow \varphi$. The derivation of φ from S can be summarized in a syntax tree for φ. The syntax tree is a tree where the internal nodes are non-terminal symbols, the leaf nodes are terminal symbols and the root node is the initial symbol. If a node $\langle n\rangle$ has child nodes with labels $\alpha_1, \alpha_2, \ldots, \alpha_k$, then there has to exist a production of the form

$$\langle n\rangle := \alpha_1\alpha_2 \ldots \alpha_k$$

If when traversing a syntax tree t of a grammar G and collecting the terminal symbols we obtain a sequence φ, then t is the syntax tree of φ w.r.t. G. The sequence φ belongs to $L(G)$ if there exists such a syntax tree t for φ. Figure 3(b) presents the syntax tree for the statements in Figure 3(a), w.r.t. the BNF grammar in Figure 2.



(a) (b)

Figure 3: (a) Part of a C-like program and (b) its corresponding syntax tree.

Pattern Syntax Trees

A syntax tree represents the syntactical structure of a program. An instrumenter can use this structure to derive the program graph, as well as to decide where to include the log statements into the program. IDeL uses this approach. Therefore, there must exist a generic way to specify, select and handle portions of the syntax tree to which the IDeL constructions can be applied. This is done with *pattern trees* and *matching*. A pattern tree denotes a set of sub-trees of a syntax tree. The matching is an operation that will identify a particular sub-tree that is denoted by a pattern tree. These two points are further discussed in the following paragraphs.

We introduce a set $\mathcal{M}$ of meta-variables and extend the syntax tree by allowing for leaves to be meta-variables as well as terminal symbols. Moreover, in this extension the root node can be any non-terminal symbol (not only the initial one, as in a syntax tree). We call this extended syntax tree *pattern tree*. Each meta-variable has an associated non-terminal symbol, which is called its type. A meta-variable can only occur where a non-terminal of its type also could. A meta-variable can be either free or bound. Every bound meta-variable is associated to a sub-tree which can be derived from its type. Figure 4 shows an example of a pattern tree for a statement in C language[1]. We prefix the meta-variables with a colon (:), as a way to distinguish from ordinary identifiers.

Patterns are specified in the following notation. The simplest pattern is formed by an anonymous meta-variable, as its root node. This pattern is expressed by the non-terminal symbol that is its root node enclosed in squared brackets. For example, [S] is a pattern whose root node is an anonymous meta-variable of type $\langle S \rangle$. In a more elaborated notation, the non-terminal root symbol is put in squared brackets, as before, but following it, in angle brackets, is included a sequence of terminal

[1] Throughout this paper, we assume a simplified grammar of C language. This simplification is carried out to make the examples self-contained and, hopefully, more understandable. The most important non-terminal symbols of this grammar are $\langle S \rangle$ (standing for statements), $\langle E \rangle$ (standing for expressions) and $\langle ID \rangle$ (standing for identifiers).

symbols and meta-variables that should be parsed to generate the pattern tree. For example, the pattern tree in Figure 4 is expressed by [S< while (:e) :s >]. Note that inside the angle brackets the grammar of the product, rather than the IDeL's grammar, is to be respected. Nonetheless, meta-variables come from IDeL itself and, thus, the previous pattern will only be valid if the meta-variables :e and :s are declared with proper types.

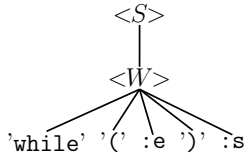


Figure 4: The pattern tree for '`while ( :e ) :s`' in a simplified grammar of C language. The types of '`:e`' and '`:s`' are $\langle E \rangle$ and $\langle S \rangle$, respectively.

For matching, we take a tree pattern p and a syntax tree t and try to unify them, using an algorithm similar to the one employed by the Prolog language [3]. A matching can either fail or succeed. In case of success, the meta-variables in the tree pattern, if any, are bound to sub-trees of t, in a way that makes them unrestrictly interchangeable. In case of failure, no meta-variable unification occurs. Figure 5 presents a successful matching for the tree in Figure 3(b) and the pattern tree in Figure 4.

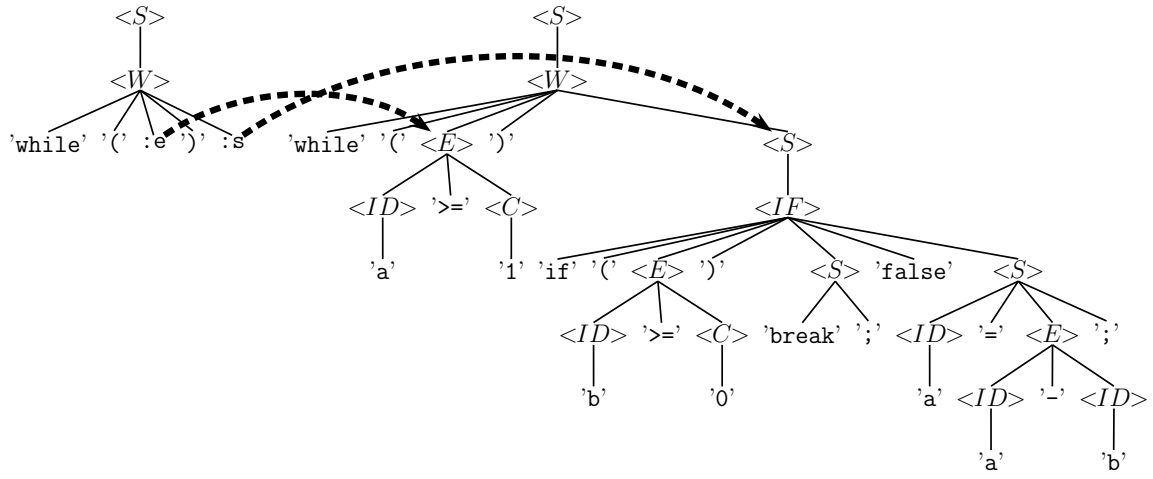


Figure 5: Matching of the tree in Figure 3(b) and the pattern tree in Figure 4.

4 IDeL: Main Features

In order to design an abstract mechanism to instrument programs, it is necessary to select a generic intermediate format, so that, for every language, the programs must be translated into this format. This mechanism must then provide methods to handle the intermediate format and to specify the

aspects relevant for the instrumentation. For example, in the approach undertaken by Chaim [5], a language — called LI — was devised. In the IDeL's approach, we decided to use syntax trees as the intermediate format. The reason for our choice is twofold. Firstly, programs written in most languages can (with more or less effort) be translated into a syntax tree and the techniques for this are well-established and well-defined in grammar theory [2]. Secondly, by using concepts from the transformational programming paradigm (exemplified by languages such as TXL [6] and Refine [10]) we can define methods for handling the syntax tree. We have already employed similar techniques in a case study with TXL [17] and in defining a language for describing and generating mutants [16].

4.1 Extended Syntax Tree

Recalling our goals, we need some mechanism to: i) derive the program graph; ii) annotate some relevant information (e.g. variable definitions and uses); and iii) insert proper statements into the program, in order to collect the relevant runtime data.

For this purpose, we extend the syntax tree with the ability to store some special data. An inner node in a syntax tree has two associated objects: a *graph node mapping* and an *insertion list*. The *graph node mapping* represents the relationships between a symbolic name and the actual graph node. For example, the graph node mapping can associate the name `$begin` to the graph node 56. The *insertion list* records what kind of statements should be inserted in order to implement the instrumentation for this particular tree node.

4.1.1 Graph Node Mapping

There is some information in an instrumentation process that depends on the context. For example, if we find a *break* statement in a C program, it is necessary to insert an edge in the program graph from the current graph node (the one in which the *break* statement occurs) to the graph node after the innermost iteration or switch statement. However, to keep IDeL simple, we have decided that the constructions of a program will be analyzed individually. Therefore, it is necessary to provide some way to make the relevant context information available when examining a particular construction.

We tackle this problem by associating graph node mappings to every tree node. In a graph node mapping, actual graph nodes can be assigned to, and retrieved from, symbolic names. The graph node mappings are arranged in a hierarchical structure, in such a way that if a particular symbolic name is not defined in the graph node mapping of a tree node, this symbolic name is recursively searched in the graph node mappings of the ancestors of this node. For example, in the case of the *break* statement mentioned above, we can assign the symbolic name `$break` to the graph node after the iteration statement being currently processed in the tree node referring to the body of the statement. Thus, whenever a *break* statement is found, one simply needs to refer to the symbolic name `$break`.

4.1.2 Insertion List

As mentioned before, one of the steps in the instrumentation process may require the insertion of log statements into program, in order to register some information about its execution. The actual

statements that are inserted depend not only on the structure of the program and the semantics of the language, but also on the purpose of the instrumentation. The insertion of new statements in the program would imply in an alteration in its syntax tree. As the syntax tree is the element that guides the whole process of instrumentation, if alterations in the syntax tree were allowed, it would be necessary to deal with them, bringing an unnecessary complexity to the semantics of IDeL. Therefore, instead of immediately inserting the log statements, the required insertions are appended to the insertion list attached to the respective tree node. After the analysis of the syntax is finished, all the insertion lists are traversed and the required insertions are made (see Section 4.3).

4.2 Instrumenter Description

An instrumenter description in IDeL is divided into three parts: unit identification, unit processing and program transformation. In this section we discuss each of these parts and illustrate the constructions of the IDeL language with examples related to an instrumenter for a simplified grammar of C language. A complete example can be found elsewhere [18]. We used the instrumentation schema presented in [11].

4.2.1 Unit Identification

In this part, it is defined what will be recognized as an instrumentation unit (e.g. a function in C or a method in Java). For each unit, a separate program graph will be generated. The units are characterized by a list of one or more patterns. The syntax tree is traversed and, if a subtree t matches any of these patterns, t is processed as a unit. Figure 6 presents the declaration of the pattern used to identify units in our example.

```
1  unit
2  var
3          :s     as [S]
4          :name  as [ID]
5          :pars  as [function_argument_list]
6          :type  as [type_specifier]
7  named by
8          :name
9  pattern
10         [function_definition < :type :name :pars :s >]
11 end unit
```

Figure 6: Unit Identification Pattern.

Lines 2-6 declare the meta-variables used in this pattern. The pattern is presented in Line 10. The declaration in Line 8 indicates that the units found will be named by the identifier unified to the meta-variable :name.

4.2.2 Unit Processing

Unit processing is the main part of an intrumenter description. It is responsible for defining how to derive the program graph and to append some special marks to the insertion list of the syntax tree nodes, when necessary. These marks indicate what kind of transformation must be made in the

syntax tree in order to insert the log statements. The unit processing part is divided into a sequence of *processing steps*. A processing step consists of the sequential application of *instrumentation rules*. An instrumentation rule defines how a particular element of the program unit must be instrumented. For instance, in our example there are instrumentation rules that define how to deal with *while* statement, *if* statements, control expressions, and so on.

An instrumentation rule is composed of seven sections: name definition, meta-variable declaration, pattern definition, graph node creation, graph topology definition, assignment and insertion marking. Among these, only the name and pattern sections are mandatory. Figure 7(a) presents a rule to instrument *while* statements. The application of this rule triggers the following tasks (not necessarily in this order):

Task 1. to create a graph node to represent the control expression of the *while*. Let *control* be this graph node.

Task 2. to insert into the graph the edges necessary to reflect the potential control flows [11]. Let *begin* and *end* be the graph nodes representing the statement that is immediately before and immediately after the *while*, respectively. Thus, the edges that must inserted are:

i) from *begin* to *control*,
ii) from *control* to the graph node representing the beginning of the body statement of the *while*,
iii) from the graph node representing the end of the body statement to *control* and
iv) from *control* to *end*.

Task 3. to set up the relevant context information:

i) every *break* statement in the body statement can refer to *end*,
ii) every *continue* statement in the body statement can refer to *control*,
iii) the uses of variables in the control expression are p-uses.

Task 4. to include marks to indicate that log statements should be inserted in the control expression, the body statement and the end of the *while*, respectively.

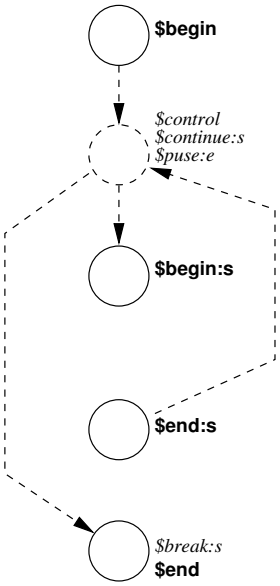
Now, we discuss the instrumentation rule in Figure 7(a), illustrating its sections and relating them to these tasks. Line 1 is the name section, which, in this case, defines the name of this rule as "While". Indeed, this name is only for documentation purpose and has no impact on the rule semantics. Lines 2 to 4 are the meta-variable declaration section, which introduces the meta-variables that may appear in the remaining of the instrumentation rule. In this example, the meta-variables :e and :s are declared with types $\langle E \rangle$ and $\langle S \rangle$, respectively.

Lines 5 and 6 define the pattern to which the sub-trees of the syntax tree must match in order for this instrumentation rule to be applied, unifying the meta-variables accordingly. In this pattern, the meta-variables :e and :s are unified to the control expression and the body of the *while*, respectively. Line 7 is the node declaration section. This section is related to Task 1. In this example, it creates a new node, adds it to the graph and assigns it to the symbolic name `$control` in graph node mapping corresponding to the matched sub-tree.

In the next three sections of this instrumentation rule, it is necessary to refer to nodes assigned to symbolic names. We use the following notation. A reference of the form `$begin` refers to the

```
1  rule While
2  var
3     :e as [E]
4     :s as [S]
5  pattern
6     [S<while ( :e ) :s>]
7  declare node $control
8  graph
9     $begin   -> $control
10    $control -> $begin:s
11    $end:s   -> $control
12    $control -> $end
13 assignment
14    assign $break:s    to $end
15    assign $continue:s to $control
16    assign $puse:e     to $control
17 instrument
18    add checkpointBefore $control to :e
19    add checkpointBefore $begin:s to :s
20    add checkpointAfter  $end     to  self
21 end rule
```

(a)



(b)

Figure 7: (a) Instrumenting a *while* statement; (b) graph alteration during the instrumentation of an *while* statement.

graph node assigned to the symbolic name "begin" in graph node mapping of the sub-tree to which the instrumentation rule is being applied. A reference of the form `$begin:s` refers to the graph node assigned to the symbolic name "begin" in graph node mapping of the sub-tree to which the meta-variable :s is unified. Any construction in the instrumentation rule that refers to a non-assigned symbolic name is ignored.

Lines 8 to 12 are the graph topology definition section, which declares graph edges and is related to Task 2. For example, Line 9 creates the edge that links the graph node assigned to `$begin` and the graph node assigned to `$control`. Line 10 creates the edge that links the graph node assigned to `$control` and the graph node assigned to `$begin` in the sub-tree unified to the meta-variable :s.

Lines 13 to 16 are the assignment section. This section is used to assign symbolic names and are related to Task 3. For example, Line 14 assigns the symbolic name `$break` of the graph node mapping of :s to the graph node assigned to `$end`. Therefore, in whichever instrumentation rule is applied to the statements in the subtree :s, any reference to the symbolic names `$continue` and `$break` will refer to the nodes assigned to, respectively, the control expression and the statement after the *while*. In line 16, we assign the symbolic name `$puse` (that stands for predictive use) of the graph node mapping of :e to the graph node assigned to `$control`. Therefore, any instrumentation rule applied to :e can refer to `$puse` (See the discussion on Figure 10).

Lines 17 to 20 are the insertion marking section. In this section, which is related to Task 4, the instrumentation rule appends special markings to the insertion lists in order to indicate where log statements must be insert. For instance, Line 18 appends a marking *checkpointBefore* in the insertion list of :e. The program transformation part (Section 4.2.3) defines how the program must

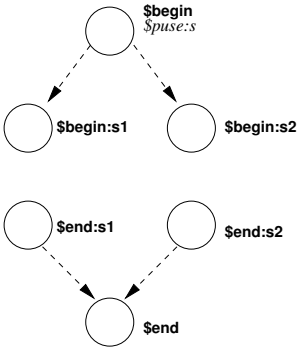
be changed to insert the statements related to this marking. The keyword `self` in line 20 refers to the matched sub-tree.

Figure 7(b) illustrates how the *While* rule works. The symbolic names `$begin`, `$begin:s`, `$end:s` and `$end` are already assigned to the suitable nodes prior to applying this rule. The node and edges that were created by this rule are highlighted with dashed lines, whereas the newly assigned symbolic names are in italics.

Figure 8(a) presents the instrumentation rule of an *If-Then-Else* statement. Note that the same overall structure is applied. Lines 9 to 12 link the graph node assigned to `$begin` in the matched sub-tree to the graph nodes assigned to `$begin` in the meta-variables :s1 and :s2 (that correspond to the *Then* and *Else* statements, respectively). Analogously the graph nodes assigned to `$end` in :s1 and :s2 are linked to graph node assigned to `$end` in the matched sub-tree. Figure 8(b) presents the corresponding graph alteration for the *IfThenElse* instrumentation rule.

```
1  rule IfThenElse
2  var
3     :e  as [E]
4     :s1 as [S]
5     :s2 as [S]
6  pattern
7     [S<if ( :e ) :s1 else :s2>]
8  graph
9     $begin  -> $begin:s1
10    $begin  -> $begin:s2
11    $end:s1 -> $end
12    $end:s2 -> $end
13 assignment
14    assign $puse:e to $begin
15 instrument
16    add checkpointBefore $begin:s1 to :s1
17    add checkpointBefore $begin:s2 to :s2
18    add checkpointAfter  $end      to self
19 end rule
```

(a)



(b)

Figure 8: (a) Instrumenting an *if-then-else* statement. (b) graph alteration during the instrumentation of an *if-then-else* statement.

Figure 9 presents the instrumentation rule of the *Break* statement. The only task that should be carried out is to link the graph node assigned to `$begin` in the matched sub-tree to the graph node assigned to `$break` (line 5). The hierarchical structure of the graph node mapping (discussed in 4.1.1) ensures that the graph node to be retrieved will be the one assigned in the closest antecessor of the matched sub-tree. Thus, the instrumenter will respect the context of the *Break* statement.

Figure 10 shows other two examples of instrumentation rules that illustrate some important features of IDeL. The instrumentation rule in Figure 10(a) deals with the assignment statements of the C language. Note that the control flows directly from the beginning to the end of the statement (line 8). Line 9 marks the information about a definition of the identifier unified to :d at the graph node `$begin` in program graph. Note that the *mark* declaration collects information about the

```
1  rule Break
2  pattern
3     [S< break ; >]
4  graph
5     $begin -> $break
6  end rule
```

Figure 9: Instrumenting a *break* statement.

program in the form of relationships between an element of the program and a node in the graph, e.g. the meta-variable :d and the node `$begin`. Line 11 assigns the symbolic name `$cuse` (that stands for computational use). Different treatments are employed w.r.t. definitions and c-uses of variables in the *Assignment* rule. While this rule marks a definition of :d, the marking of the c-uses in :e is made by the rule *Use*, shown in Figure 10(b). The reason for this distinction is that :d is an identifier, whereas :e is an expression, possibly composed of identifiers and other elements (e.g. operators).

```
1  rule Assignment
2  var
3     :d as [ID]
4     :e as [E]
5  pattern
6     [S< :d = :e ; >]
7  graph
8     $begin -> $end
9     mark definition of :d at $begin
10 assignment
11    assign $cuse:e to $begin
12 end rule
```

(a)

```
1  rule Use
2  var
3     :u as [ID]
4  pattern
5     [ID< :u >]
6  graph
7     mark puse of :u at $puse:u
8     mark cuse of :u at $cuse:u
9  end rule
```

(b)

Figure 10: (a) Variable definition handling; and (b) variable use handling

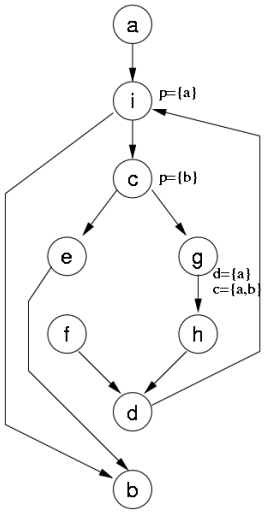
The instrumentation rule in Figure 10(b) deals with the uses of variables. Whenever an identifier is matched (line 5), the instrumenter marks a p-use and a c-use of the identifier (respectively, lines 7 and 8). Recall that if a declaration refers to a symbolic name that is not assigned, this declaration is ignored. Therefore, the p-use and c-use will be only marked if the respective symbolic names is actually defined, and, thus, the kind of use of an identifier can be controlled by assigning one or another of these symbolic names. Note that we mark the p-use in the node where the identifier occurs, and not in the edges of graph (as discussed in Section 3.1). Nevertheless, this does not reduce the expressiveness of IDeL in this case, since the edges of a p-use can be defined as the edges that originate in the graph node where the p-use was marked.

Figure 11(b) presents the program graph obtained by the application of the proper rules to the program in Figure 11(a). Every rule described in this paper was applied at least once in order to derive this graph. For example, the nodes `a`, `i`, `c`, `d` and `b` correspond to, respectively, the symbolic names `$begin`, `$control`, `$begin:s`, `$end:s` and `$end` in Figure 7(b). Note that `c` is to

the symbolic name `$begin:s` in the *While* rule, but to the symbolic name `$begin` in the *IfThenElse* rule.

```
while (a >= 1)
    if (b >= 0)
        break;
    else
        a = a - b ;
```

(a)



(b)

Figure 11: (a) Part of a C-like program and (b) the program graph (before the reduction and rearrangement).

4.2.3 Program Transformation

In the unit processing, the *instrument* declarations append markings in insertion lists of the syntax tree nodes to indicate where log statements must be insert. In the program transformation, it is defined how the program should be changed in order to insert such statements. Note that the alterations depend not only on the language of the program being instrumented, but also on the purpose of the instrumentation. It is composed by a list of *replace* declarations. For example, Figure 12(a) defines how to transform the program due to the *instrument* declaration in line 18 in Figure 7(a), which appends a *checkpointBefore* marking in the tree node of the *while* control expression. Analogously, Figure 12(b) defines how to transform the program when a *checkpointBefore* marking is found in a tree node that represents a statement.

Consider the replace declaration in Figure 12(a), which defines how to instrument an expression with a mark *checkpointBefore*. Line 5 defines the pattern that should be matched for this *replace* declaration to be applied, as well as the mark that should be in the insertion list of the matched sub-tree. Line 8 is the substitution pattern that will replace the matched sub-tree. Note that the statement check(:n) is inserted before the expression matched by :e, separated by a comma. The meta-variable :n is bound to the graph node assigned to `$node` in order to be used in the substitution pattern. Note that `$node` is not part of the target language grammar and, therefore, it cannot occur in a pattern. Thus, it is necessary to bind a meta-variable to symbolic name and, then, use this meta-variable in the pattern. Observe also that, in what concerns IDeL, the choice of check(:n) is

```
1  replace
2  var
3     :e as [E]
4     :n as [C]
5  checkpointBefore $node [E< :e >]
6  binding :n to node $node
7  by
8     [E< check(:n), (:e) >]
9  end replace
```

(a)

```
1  replace
2  var
3     :s as [S]
4     :n as [C]
5  checkpointBefore $node [S< :s >]
6  binding :n to node $node
7  by
8     [S< {
9           check(:n);
10          :s
11       } >]
12 end replace
```

(b)

Figure 12: Implementing a checkpoint *before* (a) an expression and (b) a statement, respectively.

arbitrary. IDeL only requires that a syntactically correct statement be inserted. This statement should be defined elsewhere in a suitable way, so that it will gather the appropriate information.

4.3 Applying an Instrumenter Description

The application of an IDeL instrumenter description is ruled by three elements: a program P to be instrumented (written in a language L), a context free grammar G for L and an instrumenter description (in accordance to L). The execution is accomplished in five phases:

1. **Parsing:** The program P is parsed and, if it is correct w.r.t. G, its syntax tree is built.

2. **Unit Identification:** The syntax tree is traversed in order from the root node, trying to match the patterns defined in the unit identification part of the instrumenter. Whenever a sub-tree matches one of those patterns, a new program graph is created and the matched sub-tree is processed (phase 3). After traversing the tree, the transformation phase begins (phase 4).

3. **Unit Processing:** During the application of an instrumenter description in IDeL, every processing step (as defined in Section 4.2.2) is applied, in sequence. For each node in the syntax tree, each instrumentation rule in the step is tried. If the pattern of the rule matches the tree node, the instrumentation rule is applied. After the application of the last step, the resulting graph is reduced, rearranged and output.

   (a) **Graph Reduction:** The program graph produced by processing an IDeL description may have nodes that can be merged without loosing any information about the program it represents. For example, in Figure 11(b) the graph node `h` will always be executed after the graph node `g`. To reduce the graph, an algorithm is applied in order to merge nodes that would be executed in sequence. Two nodes n_1 and n_2 are merged if n_2 is the *only* successor of n_1 and n_1 is the *only* predecessor of n_2 (i.e., n_2 is always executed just after n_1). Moreover, unreachable nodes (i.e. nodes to which there is no path from the initial node) are removed (e.g. node `f` in Figure 11(b)).

(b) **Graph Rearrangement:** The graph rearrangement algorithm chooses unique labels to the graph nodes, trying to resemble the program structure as much as possible. Figure 13 shows the program graph in Figure 11(b) after the reduction and rearrangement.

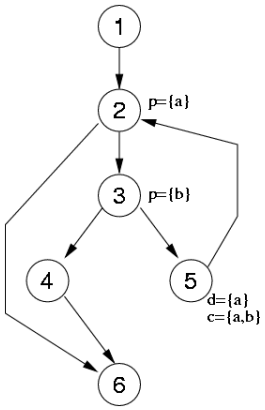


Figure 13: Reduced program graph.

(c) **Graph Output:** After the reduction and rearrangement of the graph, it can be output. Currently, the program graph (its nodes, edges and the information marked in the nodes) is output in XML format. Alternatively, it can be output in `dot` digraph format of GraphViz [8]. GraphViz is a powerful tool for drawing graphs and is able to generate several different kinds of image format (e.g., JPEG, PNG, EPS).

After outputting the graph, the instrumenter resumes phase 2, searching for another unit.

4. **Unit Transformation:** In this phase, the syntax tree is traversed, starting from the root node, verifying in the insertion lists of its sub-trees whether it is necessary to insert log commands. When an insertion is requested, the *replace* declarations are inspected until identifying one that is applicable for the respective sub-tree and insertion type. If an applicable *replace* declaration is found, the syntax tree is accordingly changed and the tree traversal goes on.

5. **Unparsing:** The syntax tree (eventually changed in the previous phase) is traversed and every terminal symbol is collected. The sequence of terminal symbols obtained is the instrumented program. Considering the sample program in Figure 11(a), the instrumentation process will produce the instrumented program presented in Figure 14. As discussed previously, the statement *check* was arbitrarily chosen as being a statement to register the execution of each node. It can be noted that, for keeping the program syntactically correct w.r.t. the C grammar, it was necessary to include extra open/close curling brackets around the statements.

4.4 Reusing Instrumenter Descriptions

IDeL language was designed so that an instrumenter description could be changed as easily as possible, either to include new features or to change existing ones. This design increases the potential reuse of an instrumenter description for other similar goals. For example, the declarations that impact on the graph topology can be changed without, in principle, worrying about how the

```
{
  check(1);
  while (check(2), a >= 1)
    {
      check(3);
      if (b >= 0)
      {
        check(4);
        break;
      }
      else
      {
        check(5);
        a = a - b ;
      }
    }
  check(6);
}
```

Figure 14: Instrumentated program.

source program should be transformed in order to log its execution. Conversely, it is possible to change the way the program is transformed without any impact on the graph topology.

To illustrate the reuse of an instrumenter description, consider a situation where, besides logging the fact that an *if* statement with a relational operation was executed, one wants to log i) the values used in the control expression; and ii) the results of the expression. This example was inspired in the instrumenter description proposed by Bueno and Jino [4] for deriving program test cases with genetic algorithms. In order to specialize the instrumenter to behave in this way, it is only necessary to include new *replace* declarations to customize the handling of *if* statements, in contrast to the declaration in Figure 12(b), which handles any statement. These more specialized declarations should be placed before the general one in the instrumenter description, so that they will be found and applied first. Figure 15 presents an example of the *replace* declaration related to the relational operator '`>=`'. Observe that, in this case, the addition of a new functionality only requires a change in a delimited location, keeping all the remaining description unchanged. Figure 16 presents the instrumented program that would result from the application of this changed instrumenter description to the program in Figure 11(a).

5 Operational Aspects

We have developed a system, named `idelgen` (standing for IDeL **Gen**erator), to support the application of an instrumenter description. In order to use IDeL to describe an instrumenter for programs in a language L, it is necessary to provide a context-free grammar G for L. Grammar G will be used to check whether both the programs and the patterns in the instrumenter description are syntactically correct. Given G, `idelgen` will produce a program called `idel.G`. In its turn, this

```
replace
var
   :i as [ID]
   :c as [C]
   :s1 as [S]
   :s2 as [S]
checkpointBefore $node
   [S< if (:i >= :c) :s1 else :s2 >]
binding :n to node $node
by
   [S< {
         check(:n);
         if (:i >= :c) {
           log_value(:n, (:i), (:c), GE);
           :s1
         } else {
           log_value(:n, (:i), (:c), LT);
           :s2
         }
       } >]
end replace
```

Figure 15: Implementing a checkpoint *before* an *if* statement.

```
{
  check(1);
  while (check(2), a >= 1)
    {
      check(3);
      if (b >= 0)
      {
        log_value(3, (b), (0), GE);
        check(4);
        break;
      }
      else
      {
        log_value(3, (b), (0), LT);
        check(5);
        a = a - b ;
      }
    }
  check(6);
}
```

Figure 16: New instrumentated program.

program can then be run with an instrumenter description ID and a program P, producing the respective instrumented program and graphs (one for each program unit).

In order to manipulate G and generate `idel.G`, `idelgen` uses the tools `bison` and `flex`, which are open source programs similar to, respectively, `yacc` and `lex` [12]. Although these tools ease the

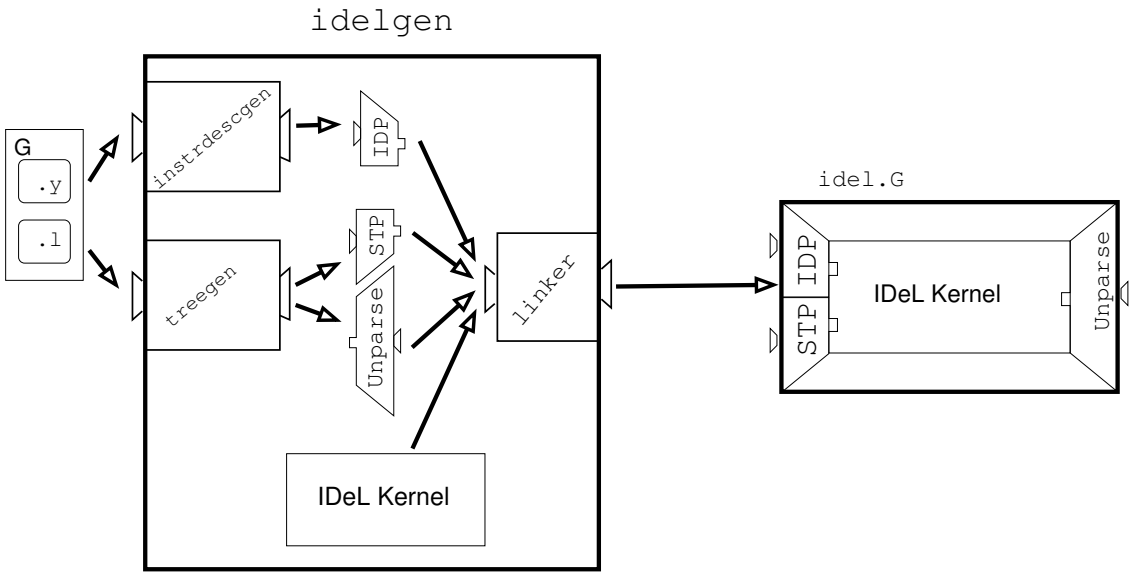


Figure 17: `idelgen` Execution Schema

task of manipulating grammars, they, on the other hand, restrict the set of grammars that `idelgen` can currently deal with to LALR(1) grammars [2, 12, 15]. G is input to `idelgen` in two files: `G.y` and `G.l`. File `G.y` is a set of grammar rules, written in a subset of `yacc` syntax [12]. File `G.l` contains the rules to lexically analyze L, defining the actual form of the terminal symbols of G. It is a subset of the `lex` syntax [12]. Indeed, these files can be thought of as minimal standard `yacc` and `lex` files, from which all the so-called semantic actions were stripped off.

The program `idel.G`, generated by `idelgen` from G, can be divided in two parts: one with the elements that depend on G and another with elements that do not. Figure 17 depicts how these parts interact and illustrates the overall schema by which `idelgen` builds `idel.G`. The part depending on G is handled by three modules of `idelgen`: `treegen`, `instrdescgen` and `linker`. The grammar independent part is embodied in the so-called IDeL Kernel, which is responsible for interpreting the instrumenter description and manipulating the syntax tree accordingly.

Module `treegen` analyzes G and generates two components:

(i) `STP` (Syntax Tree Processor), which is responsible for syntactically analyzing a source program P w.r.t. G and to generate the syntax tree; and

(ii) `Unparse`, which is responsible for converting the resulting syntax tree into the instrumented program.

Module `instrdescgen` analyzes G and generates the component `IDP` (Instrumenter Description Processor), which analyzes an instrumenter description ID w.r.t. G and generates an internal abstract representation of it. Finally, the `linker` module will link all these grammar-depending components and the IDeL Kernel and generate the program `idel.G`. The program `idel.G` can then be used to instrument a source program P w.r.t. an instrumenter description ID (Figure 18). Both P and ID are input to `idel.G` and internally processed by `STP` and `IDP`, respectively Then, IDeL

Kernel will apply ID, generating one or more program graphs (one for each unit). The resulting syntax tree is processed by `Unparser` in order to generate the actual instrumented program P'.

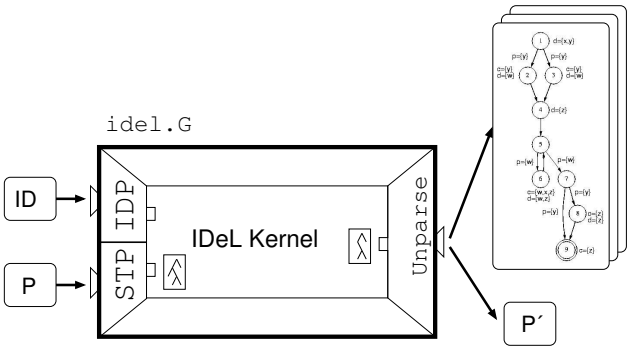


Figure 18: `idel.G` Execution Schema

6 Concluding Remarks

Program instrumentation is a technique largely employed in software engineering, suitable to obtain an abstract view of an program, as well as to inspect its execution. The instrumentation is usually conducted with a specific purpose in mind, such as testing coverage assessment and program visualization. In this paper we presented a language for describing instrumenters. This language abstracts and captures the most important concepts of the instrumentation process. It embodies these concepts by providing simple constructions to determine how to derive the program graph, collect important data about the program and insert log statements to register the program execution.

Currently, IDeL does not provide mechanisms to cope with *goto* statements. Actually, a *goto* statement demands that nodes be referred to by means of label identifiers. This point can be tackled by including a global lookup table of identifiers. However, we think that a *goto* statement will only be a special case of reference to some element of the program and we are studying how to generalize the construction to be useful for more general cases, such as referring to entry and return nodes of an inter-procedural call [9].

The primary motivation for the development of IDeL (and its primary use) is within a larger project we are undertaking to devise a generic testing tool; generic in the sense that the same tool may be used for different languages, namely, C, C++ and Java. This generic tool will be a framework with built-in features for the common tasks that should be made by a testing tool. The framework should provide some way to describe the specific characteristics. IDeL will be used for describing how the instrumentation takes place for a specific language. Note that the instrumenter description requires that just the relevant parts of the grammar be considered. Therefore, even the description (or, at least, part of it) can be reused, by carefully constructing the grammar (e.g. choosing nonterminal names consistently) and writing the description (i.e. isolating features that are particular of a given language).

References

[1] Agrawal, H., Alberi, J. L., Horgan, J. R., Li, J. J., London, S., Wong, W. E., Ghosh, S., and Wilde, N. (1998). Mining system tests to aid software maintenance. *IEEE Computer*, 31(7):64–73.

[2] Aho, A. V., Sethi, R., and Ullman, J. D. (1985). *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass.

[3] Bratko, I. (1990). *Prolog Programming for Artificial Intelligence*. Addison-Wesley, Wokingham, England ; Reading, Mass., 2 edition.

[4] Bueno, P. M. S. and Jino, M. (2001). Automated test data generation for program paths using genetic algorithms. In *13th International Conference on Software Engineering & Knowledge Engineering — SEKE'2001*, pages 2–9, Buenos Aires, Argentina.

[5] Chaim, M. L. (1991). Poke-Tool — uma ferramenta para suporte ao teste estrutural de programas baseados em análise de fluxo de dados. Master's thesis, DCA/FEEC/UNICAMP, Campinas, SP.

[6] Cordy, J. R., , Carmichael, I. H., and Halliday, R. (1995). The TXL programming language — version 8. Technical report, Department of Computing and Information Science, Queen's University, Kingston, Canada.

[7] DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41.

[8] Gansner, E. R. and North, S. C. (2001). An open graph visualization system and its applications to software engineering. *Software — Practice & Experience*, 30(11):1203–1233.

[9] Harrold, M. J. and Soffa, M. L. (1991). Selecting and using data for integration testing. *IEEE Software*, pages 58–65.

[10] Kotik, G. B. and Markosian, L. Z. (1989). Automating software analysis and testing using a program transformation system. In *Proceedings of the ACM SIGSOFT'89 Third Symposium on Software Testing, Analysis, and Verification*, pages 75–84.

[11] Maldonado, J. C. (1991). *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. PhD thesis, DCA/FEEC/UNICAMP, Campinas, SP.

[12] Mason, T. and Brown, D. (1990). *Lex & Yacc*. O'Reilly.

[13] Neighbors, J. (1984). The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5):564–574.

[14] Rapps, S. and Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375.

[15] Salomaa, A. (1973). *Formal Languages*. Academic Press, New York.

[16] Simão, A. S. and Maldonado, J. C. (2001). MuDeL: A language and a system for describing and generating mutants. In *Anais do XV Simpósio Brasileiro de Engenharia de Software*, pages 240–255, Rio de Janeiro, Brasil.

[17] Simão, A. S., Sugeta, T., Maldonado, J. C., and Monard, M. C. (2001). Prolog & TXL: um estudo de caso para prototipação de ferramentas de apoio para o teste estrutural. In *Anais das 1ªs Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento*, pages

15–22, Buenos Aires, Argentina.

[18] Simão, A. S., Vincenzi, A. M. R., Maldonado, J. C., and Santana, A. C. L. (2002). Software product instrumentation description. Technical Report 157, ICMC/USP, São Carlos, SP, Brazil.

[19] Vladimir, D. (1989). *Formal Languages and Automata Theory.* Computer Science Press.