UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

# Software Product Instrumentation Description

Adenilso da Silva Simão
Auri Marcelo Rizzo Vincenzi
José Carlos Maldonado
Antonio Carlos Lima de Santana

Nº 157

RELATÓRIOS TÉCNICOS

São Carlos - SP

Instituto de Ciências Matemáticas e de Computação

ISSN - 0103-2569

# Software Product Instrumentation Description

Adenilso da Silva Simão
Auri Marcelo Rizzo Vincenzi
José Carlos Maldonado
Antonio Carlos Lima de Santana

Nº 157

RELATÓRIOS TÉCNICOS DO ICMC

São Carlos
Mar/2002

# Software Product Instrumentation Description*

**Adenilso da Silva Simão**
**Auri Marcelo Rizzo Vincenzi**
**José Carlos Maldonado**
**Antonio Carlos Lima de Santana**

Departamento de Ciências de Computação e Estatística
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo – Campus de São Carlos
Laboratório de Engenharia de Software
Caixa Postal 668, 13560-970 – São Carlos, SP, Brasil
e-mail: {adenilso, auri, jcmaldon, santana}@icmc.sc.usp.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. From an abstract viewpoint, instrumenting a software product can be divided into two main tasks: deriving the software product structure and including statements for checking runtime/simulation information.

Most instrumentation approaches are domain and/or language specific, which makes it difficult to reuse and evolve the related products. In this technical report is proposed an instrumentation oriented meta-language, named IDeL, designed for supporting the description of both tasks of instrumentation process: the product structure derivation and the inclusion of the instrumentation statements. This meta-language is to be instantiated by furnishing a context-free grammar for a specific language. To promote its practical use, IDeL is also supported by a system, name IDeLgen, that can be thought of as an application generator tailored to the instrumenting process, easing the reuse and evolution of the instrumenter. Our primary motivation in developing both IDeL and IDeLgen is to integrate them in a generic data flow based testing environment. We illustrate the main ideas in this technical report with examples from this context.

**Keywords:** Software Testing, Instrumentation, Transformational Paradigms

*Acknowledgement: We would like to thank the partial financial support of FAPESP, CNPq and Telcordia (USA). We are specially thankful to Susan Linkman for reviewing an early version of this document.

# Contents

# List of Figures

Chapter

1

# Introduction

There are several activities in software engineering that demand a program be analyzed, such as reverse engineering, performance analysis, program visualization, debugging, testing and so on The program is often scanned and the useful information is retrieved. In most cases, we look for program control and data flow structure. Hence, the program (or a function thereof) is abstracted as a graph whose nodes and edges are, respectively, the program's statements and control flow branches. The relevant information of the program is attached either to the nodes or the edges. Usually, the program graph is more suitable for manipulations and analyses.

As well as being able to derive the program graph (which is the program's static aspect), it is often also necessary to get some insight of the program execution. One may want to know what statements were executed and in what order. Mapping to the program graph, one may want to know what paths (i.e. sequences of nodes and edges) were traversed. For this purpose, the program is annotated with some special statements that do not change the program semantics but do register some information in a trace file. These two tasks together — program graph derivation and program decoration — are often referred to as *program instrumentation.* Program instrumentation, in one form or another, is present in most testing coverage criteria. For instance, the all-statement coverage, which demands every statement in the program be executed at least once, is often accomplished by instrumenting the program and analyzing the trace data.

Being a lengthy and error-prone process, program instrumentation is usually performed by an instrumenter, which is built for automating this process. It analyzes the

program according to the grammar of its language, recognizes the relevant structures of the program, and derives both the graph and the decorated program based upon the semantics of the language. An approach would be the development of software specific for instrumenting code in that language. When building instrumenters, some techniques employed in compiler construction are useful. For example, the analysis of the program structure can be accomplished with compiling tools like `yacc` and `lex` (Mason and Brown, 1990). The drawback of this approach is that, although the main ideas remain the same, each new language would demand the development of new software. Another approach would be the use of transformational systems, like TXL (Cordy et al., 1995) and Draco (Neighbors, 1984). However, neither compiling tools nor transformational systems are tailored to instrumentation and one must incorporate statements and data structures to achieve this goal.

There are some features that are common to all instrumenter regardless of language and purpose. However, there are also some characteristics that are specific for a particular language or purpose. Therefore, there should ideally exist some approach for describing only the intrinsic characteristics of an instrumenter.

In this technical report, we propose a system to tackle instrumentation by providing a language that embodies constructions suitable for describing the features of the intended instrumenter. Our approach can be thought of as an *instrumenter generator*, based on transformational paradigm and graph theory and restricted to the program instrumentation domain. It aims at achieving a good trade-off between generality and complexity. The system is formed by an **I**nstrumentation **De**scription **L**anguage (named IDeL) and a "compiler" (named `IDeLgen`, standing for IDeL **Gen**erator) for actually performing the instrumentation.

IDeL is part of a larger effort we are endeavoring to specify, design and implement a generic, multi-language tool for supporting Mutation Testing (DeMillo et al., 1978) and control/data flow testing criteria (Maldonado, 1991; Rapps and Weyuker, 1985). Our work involves defining a general development framework and mechanisms to instantiate it for a particular purpose. Therefore, our primary motivation in developing IDeL was to provide the basis of these mechanisms for instantiating the tool for testing criteria analyses, therefore enabling us to derive def-use program graphs (Rapps and Weyuker, 1985) and decorate programs in order to trace their executions. So far, we have been able to use IDeL in some experiments, such as C, C++ and Java program instrumentation for testing coverage analysis and program visualization.

This technical report is organized as follows: In Section 2 we present some work related to our approach. In Section 3 we introduce some background concepts of program

graphs and grammar theories that are required for the discussion in the remaining of the article. In Section 4 we present our proposal. In Section 4.1 we describe the main features of IDeL. In Section 4.2 we describe IDeLgen, a supporting system for generating instrumenters based on an IDeL description. In Section 4.3 we discuss some results we have so far achieved. In Section 5 we make some concluding remarks and point some future work.

Chapter

2

# Related Work

There are several applications that use instrumentation as a mechanism to abstract programs in order to ease program analyses. Instrumentation is also used to trace program execution. In this section, we discuss some of the work related to instrumentation.

Rapps and Weyuker define their data flow criteria (Rapps and Weyuker, 1985) based on the def-use graph, a program graph enhanced with information about the definition (assignment) and use of variables. This information is attached to the nodes and the edges of the program graph. Then, the def-use graph is analyzed and testing requirements are derived. To verify whether these requirements were fulfilled by a test set, the program is decorated with statements to store the program flow.

In (Maldonado, 1991), Maldonado proposes data flow criteria that are based on *potential* use of variables and also use program decoration and a graph similar to def-use graph. These criteria were implemented in the tool Poke-tool (Chaim, 1991). We used the instrumentation schema presented in (Maldonado, 1991) as a guide to construct an instrumenter for C using IDeL.

Def-use graphs can also be used for analyzing data flow anomalies in a compiler. Indeed, most good compilers include mechanisms for deriving the data flow and verifying anomalies, such as unreachable statements, uninitialized variable uses, and unused computations. Def-use graphs can also be used as part of the optimizing pass. Also, the def-use information collected during the testing activity is useful in maintenance, reducing the cost of (re-)testing unaffected parts of the code.

In (Bueno and Jino, 2001), Bueno uses program instrumentation in an approach for employing genetic algorithms to generate test cases. In this work, besides information about the number of nodes, definitions and uses of variables, the fitness function of a genetic algorithm requires other information, i.e. the value of some expression, which is related with a predicative node, considered after executing with some test case.

There are cases in which either program decoration or graph derivation is required. For example, program instrumentation for performance analyses only include statements into the program that might collect information about the dynamic aspects of a particular execution, allowing the identification of performance problems, such as bottlenecks. For this case in particular, the program graph is sometimes not necessary and only program decoration is used. On the other hand, the abstract viewpoint provided by the program graph is often more appropriate for tasks aimed at obtaining an overall insight into the program structure, e.g. reverse engineering. For this case, the program does not need to be decorated.

Therefore, program instrumentation is important, being related to several aspects of software engineering (in particular, VV&T activities). In general, different types of instrumentation are required according to the objective pursed. In this way, an instrumentation tool should be flexible and, at the same time, suitably tailorable, in order to require the minimum effort.

Chapter

3

# Basic Concepts

In this section we present some background concepts related to program graphs and grammars.

## 3.1 Program def-use Graphs

After analyzing a program, the program structure is summarized in a *program graph.* The program graph topology reflects the possible flow of control in an actual execution of the program. The graph nodes represent every relevant component of the program. Usually, those components are the executable statements. Nevertheless, other components can be present, such as, declarations and function definitions. Graph edges are the relationships between the components of the program. Usually, when nodes are executable statements, the edges are the flows of control amongst the statements. An edge from node $a$ to node $b$ means that $b$ may at some point be executed after $a$ in some executions of the program. Hereafter, we will refer only to the case where the nodes are statements and the edges are flows of control. However, the ideas presented are not limited to this specific situation.

To completely represent the program, not every statement needs to be represented by a different node. For example, there are groups of statements that share the property that, in every normal execution of the program, when one of them (the first) is executed, all the others will also be executed in sequence.

In other words, there is no control flow branch from or to the middle of the group. Such a group is called a *statement block*. A program graph usually represents the statement blocks of a program and the relationship amongst them.

An extension of the program graph which is often used, specially in some testing coverage criteria (e.g. (Maldonado, 1991; Rapps and Weyuker, 1985)), is the def-use graph. A def-use graph includes information about the variables that are defined (i.e. assigned a value to) or used. There are three kinds of information:

**definitions** When values are assigned to variables.

**c-use** (Computational Use) When values of variables are used to contribute in some computation.

**p-use** (Predicative Use) When values of variables are used in an expression that controls which branch is to be followed (i.e. in a predicate expression in an *if* statement).

To explain the concepts presented above, Figure 3.1 presents a simple program (composed of a single function that returns $x^y$) and its corresponding def-use graph. The number of the nodes are written in comments in the same line of the statement(s) it represents. We can observe, for example, that node 6 is a block statement composed of two statements, since they are always executed in sequence. The graph also includes information about the definitions, p-uses and c-uses, abbreviated to `d`, `p` and `c`, respectively. For example, a p-use of the `p` variable in the control expression of the *while* statement (node 5) is represented by two labels near the edges $(5, 6)$ and $(5, 7)$ (the possible control flow branches).

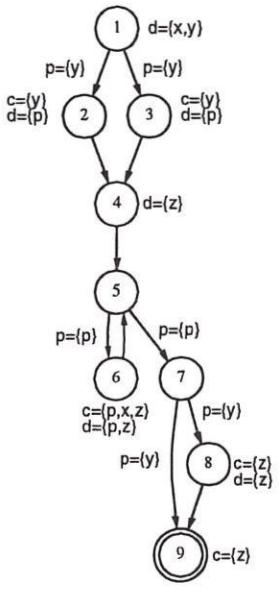
## 3.2 Grammars and Syntax Trees

In this section we present a brief introduction to grammar and language theories, needed for the discussion that follows. A thorough presentation can be found elsewhere (Salomaa, 1973). Syntax grammars are finite devices to describe usually 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 or computer 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 computer languages. Moreover, the

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

(a)




(b)

Figure 3.1: (a) A simple function that calculates $z = x^y$ and (b) its corresponding program graph.

algorithms to recognize them are computationally tractable. Context-free grammars are usually described in BNF (Vladimir, 1989). 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 $R \subseteq N \times (N \cup T)^*$ are the production rules. Rather informally, a production rule of the form $(n, \alpha)$ states that the non-terminal symbol $n$ (the *lefthand* symbol) can be replaced by the sequence $\alpha$ (the *righthand* symbol sequence) of terminal and non-terminal symbols without "inflicting" the grammar. Usually, the non-terminal symbols are expressed between angle brackets and the terminal symbols between single quotes. A production $(n, \alpha)$ is represented in the form of

$$\langle n \rangle ::= \alpha$$

Using these conversions, BNF grammars can be (and often are) described solely by means of productions, taking the terminal and non-terminal sets directly from them, and having the initial symbol as the lefthand symbol of the first production.

As an example of a BNF grammar, Figure 3.2 presents a grammar that defines a simple language. This language includes *if* and *while* statements[1]. Following the above conventions, we can derive the remaining elements of the grammar. So, we have the non-terminal set $N = \{\langle S\rangle, \langle W\rangle, \langle IF\rangle, \langle E\rangle, \langle ID\rangle, \langle C\rangle\}$, the terminal set $T = \{$'`break`', '`while`', '`if`', '`else`', '`(`', '`)`', '`;`', '`+`', '`>=`', '`=`', '`,`'$\}$, and the initial symbol $S = \langle S\rangle$. 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`' '`;`'

$\langle S\rangle$ ::= $\langle ID\rangle$ '`=`' $\langle E\rangle$ '`;`'

$\langle W\rangle$ ::= '`while`' '`(`' $\langle E\rangle$ '`)`' $\langle S\rangle$

$\langle IF\rangle$ ::= '`if`' '`(`' $\langle E\rangle$ '`)`' $\langle S\rangle$ '`else`' $\langle S\rangle$

$\langle E\rangle$ ::= $\langle ID\rangle$ '`>=`' $\langle C\rangle$

$\langle E\rangle$ ::= $\langle ID\rangle$ '`+`' $\langle ID\rangle$

$\langle E\rangle$ ::= $\langle ID\rangle$ '`(`' $\langle E\rangle$ '`)`'

$\langle E\rangle$ ::= $\langle E\rangle$ '`,`' $\langle E\rangle$

$\langle ID\rangle$ ::= some identifier

$\langle C\rangle$ ::= some integer

Figure 3.2: 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 $(n, \alpha)$. 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 $\varphi$ from $S$ can be summarized in a syntax tree for $\varphi$. The syntax tree is a tree where the internal nodes are non-terminal

[1]We adopted this artificial, truncated language in order to be able to exemplify the concepts in the remaining section. This language can be thought of as a sub-set of the C programming language. The simple statements in this language resembles the corresponding statements in C language. We will deliberately assume the C semantics for them.

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 $\varphi$, then $t$ is the syntax tree of $\varphi$ with respect to (w.r.t.) $G$. The sequence $\varphi$ belongs to $L(G)$ if there exists such a syntax tree $t$ for $\varphi$. Figure 3.3(b) presents the syntax tree for the statements in Figure 3.3(a), w.r.t. the BNF grammar in Figure 3.2.

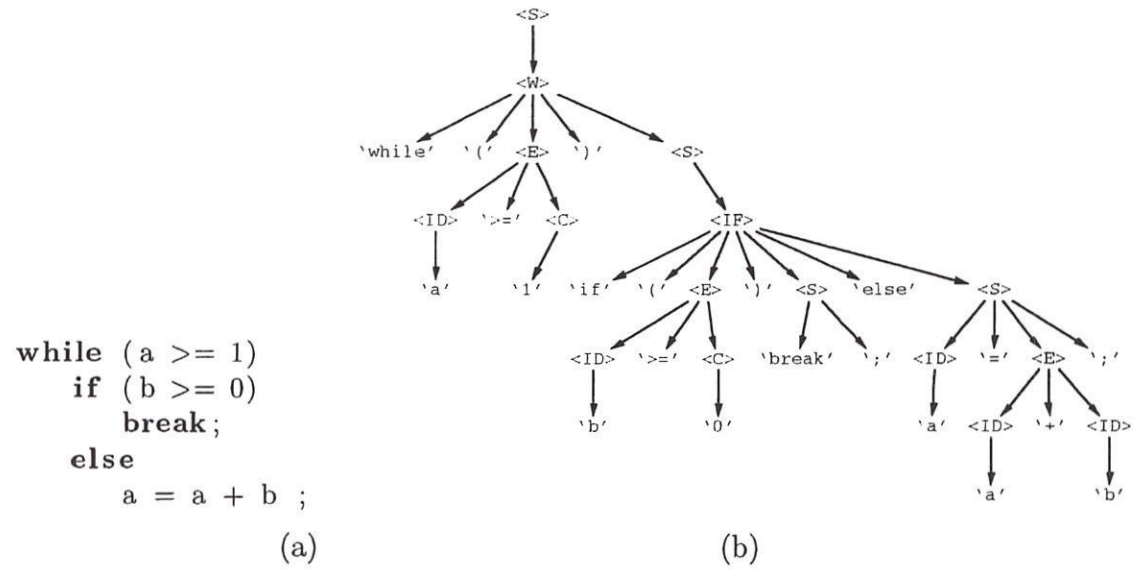



Figure 3.3: (a) Part of a C-like program and(b) its corresponding syntax tree.

## 3.3 Pattern Syntax Trees

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 be either free or bound. Every bound meta-variable is associated to a sub-tree that can be generated from its type. Therefore, a syntax tree is just a special kind of pattern tree: a kind where every meta-variable (if any) is bound. Figure 3.4 shows an example of a pattern tree. As

a way to distinguish from ordinary identifiers, we prefix the meta-variables with a colon (:). Even in the presence of meta-variables, the children of a node must be in accordance with its productions, i.e., a meta-variable can only occur where a non-terminal of its type also could.

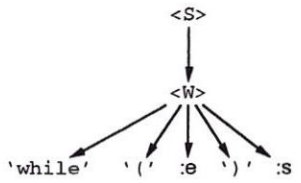



Figure 3.4: The pattern tree for '`while ( :e ) :s`'. The types of '`:e`' and '`:s`' are $\langle E \rangle$ and $\langle S \rangle$, respectively.

To specify patterns we use 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] a pattern whose root node is an anonymous meta-variable of type $\langle S \rangle$.

In a more elaborated pattern denotation, the non-terminal root symbol is put in squared brackets, as before, but following it, in angle brackets, is included a sequence of terminal symbols and meta-variables that should be parsed to generate the pattern tree. For example, the pattern tree in Figure 3.4 is denoted 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.

## 3.4 Matching

For matching, we take two tree patterns $c$ and $m$ and try to unify them, using the same algorithm as the Prolog language (Bratko, 1990). A matching can either fail or succeed. In the case of success, the meta-variables in the tree pattern are unified either to closed tree patterns or to other meta-variables, in a way that makes them unrestrictly interchangeable. In the case of failure, no meta-variable unification occurs.

Chapter

4

# IDeL & IDeLgen

## 4.1 IDeL: Main Features

In order to design an abstract mechanism to instrument programs, one must choose a general intermediate format and, for every language, the programs are translated into this format. This mechanism must then provide methods to handle the intermediate format and specify the aspects relevant for the instrumentation. For example, in the approach undertaken by Chaim (Chaim, 1991), a suitable 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:

First, programs written in most languages can (with more or less effort) be converted into a syntax tree and the techniques for this are well-established and well-defined in grammar theory (Aho et al., 1985). We can use a context free grammar for the language to drive this conversion.

Second, by using concepts from the transformational programming paradigm (and languages such as TXL (Cordy et al., 1995) and Refine (Kotik and Markosian, 1989)) we can define methods for handling the syntax tree. We have already employed similar techniques in a case study with TXL (Simão et al., 2001) and in defining a language for describing and generating mutants (Simão and Maldonado, 2001).

### 4.1.1 Enriched 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) decorate the program in the proper manner for collecting 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: the node mapping and the implementation table. *Node mapping* represents a relationship between symbolic and actual node names. For example, the node mapping can define that, say, the node referred to as `$begin` is 56. The *implementation table* records what kind of instrumenting decoration should be taken in order to implement the instrumentation for this particular tree node. For example, the implementation table can define that a checkpoint is to be added before a specific tree node.

#### 4.1.1.1 Node Mapping

There is some information in an instrumentation process that must be gathered from the context. For example, when one finds a *break* statement in a C program, one must include an edge in the graph from the current node to the 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, we need some way to let context information be available when examining a particular construction.

We tackle this problem by associating node mappings to every tree node. In node mapping, actual graph nodes can be assigned to, and retrieved from, symbolic names. The node mappings are arranged in a hierarchical structure, in that if a tree node does not map a particular symbolic name, the tree node's ancestors are recursively queried for that particular symbolic name.

For example, in the case of the *break* statement referred to above, we can assign the symbolic name `$break` to the node after the iteration statement being currently processed in the tree node referring to the body. Whenever a break statement is found, one simply needs to refer to the symbolic name `$break`. Then, by the semantics of hierarchical structure previously described, the innermost tree node that has a node assigned to this symbolic name will be retrieved.

#### 4.1.1.2 Implementation Table

Depending upon the kind of instrumentation required, it may be necessary to alter the program in order to log its execution. The alteration depends on the structures of the program and the language semantics. Most of the time, the alteration is made either before or after a component and is related to a particular node. Therefore, every tree node has a table recording the alterations that must be made to a component. As the last phase in the instrumentation, the tree is traversed and every alteration is realized (see Section 4.3).

### 4.1.2 Overall Structure

The instrumentation in IDeL is divided into three main parts: unit identification, unit processing and implementation. In this section we discuss these parts. To introduce the constructions of the IDeL language, we present some examples related to an instrumenter for the simple language in Figure 3.2, shown in Section 3.2.

#### 4.1.2.1 Unit Identification Part

The unit identification part will find the toplevel units (w.r.t. to this instrumentation), such as functions in C. For every toplevel unit a separate program graph will be generated. In the instrumenter, the units are characterized by a list of patterns. Whenever one of these patterns is found, the unit (i.e. the tree matched to the pattern) is processed.

#### 4.1.2.2 Unit Processing Part

The unit processing part is the kernel of the IDeL's approach and is responsible for deriving the program graph and annotating in the tree the points where the implementation part should include checkpoints. The unit processing part is divided into a sequence of *processing steps*. Each processing step will be applied to the unit in sequence.

A processing step consists of the sequential application of *instrumentation patterns*. An instrumentation pattern is composed of several sections: name section, meta-variable declaration section, match section, node declaration section, graph topology section, assignment section and instrument section. Only the name and the match sections are mandatory.

Figure 4.1 presents an example of an instrumentation pattern. This pattern instruments *while* statements in our simple language. Assuming the usual semantics of a *while* statement, the tasks an instrumenter should perform are (not necessarily in this order):

```
1  pattern While
2  var
3      :e as [E]
4      :s as [S]
5  match
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 checkpoint $control before :e
19     add checkpoint $begin:s before :s
20     add checkpoint $end     after  self
21 end pattern
```

Figure 4.1: Instrumenting a *While* Statement.

**Task 1.** to create a node for the control expression.

**Task 2.** to link the nodes to represent the control flows. The possible control flows are i) from the statement before the *while* to the control expression, ii) from the control expression to the beginning of the body statement, iii) from the end of the body statement to the control expression and iv) from the control expression to statement after the *while*.

**Task 3.** to set some context information, so that, i) when a break statement is found in the body statement, the node after the *while* will be used, ii) when a continue statement is found, the control expression will be used and iii) the uses of variables in the control expression are p-uses.

**Task 4.** to include checkpoints to log the execution of the control expression, the body statement and the end of the loop.

Next, we discuss the instrumentation pattern in Figure 4.1, illustrating its sections and relating them to tasks that are necessary for instrumenting a *while* statement. Line 1

marks the name section ("While", in this case). It declares the name of the pattern. Indeed, this name is only for documentation purpose and has no impact on the pattern semantics. Lines 2 to 4 are the meta-variable declaration section. This section introduces meta-variables that may appear throughout this pattern. In this example, meta-variables :e and :s are declared with types $\langle E \rangle$ and $\langle S \rangle$, respectively.

Lines 5 and 6 state the pattern that is to be matched in order for this instrumentation pattern to be applied. After being matched, the meta-variables are unified accordingly. Hereafter, we refer to the matched tree node as the node in the tree that matched the pattern.

Line 7 is the node declaration section. This section is employed to create nodes and is related to Task 1. A new node is created and added to the graph. Actually, this is the only way a graph node is created. The new node is then assigned the symbolic name presented in the declaration and attached to the matched tree node (`$control`, in this case).

In the remaining three sections, it is necessary to refer to nodes assigned to symbolic names. We use the following convention. A reference of the form `$begin:s` refers to the node assigned to the symbolic name "begin" in the tree node the meta-variable :s is unified to. If the meta-variable is not indicated (as in `$begin`), then the symbolic name is looked for in the matched tree node. Whenever a symbolic name is not assigned in a tree node (nor in an ancestor), a null node is retrieved. If any of the nodes referred to in a construction is the null node, this construction is ignored.

To understand how (and why) the instrumentation works in this example, it is necessary to first explain some things that are assumed in this pattern. Every statement has two symbolic names: `$begin` and `$end`. These are, respectively, the nodes just before and just after the statement. Actually, these assumed symbolic names are provided by previous processing steps (not included in this technical report) responsible for creating and assigning them.

Lines 8 to 12 are the graph topology declaration. This is used to declare the graph edges and is related to Task 2. For example, Line 9 creates the edge that links the `$begin` and the `$control` nodes of the matched tree node. Line 10 creates the edge that links the `$control` of the matched node tree and the `$begin` node of the tree node 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 `$break` in the tree node unified to :s to the graph node assigned to the symbolic name `$end` of the matched tree node. Intuitively, this means that when the instrumenter is analyzing the

statements in the subtree :s, a reference to the "break" node will retrieve the node after the *while*. This is the usual meaning of a break statement in C language. Note also that the "continue" node will be the control node (line 15). In line 16, we assign the symbolic name `$puse` that rules the identification of variable uses (predicative uses, in this case).

The instrument section is in lines 17 to 20. It declares what kind of instrumentation is to be applied in order to decorate the program and is related to Task 4. Line 18 defines that a checkpoint should be included before the control expression :e. In the implementation part (Section 4.2.3), these declarations are converted into the actual forms. The keyword `self` in the checkpoint declaration in line 20 refers to the matched tree node.

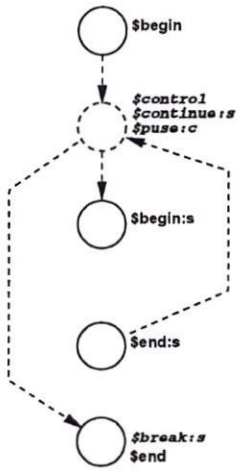



Figure 4.2: Graph Alteration during the Instrumentation of an *While* statement.

Figure 4.2 illustrates how the *While* pattern works. As previously stated, the nodes `$begin`, `$begin:s`, `$end:s` and `$end` are already assigned prior to applying this pattern. The node and edges that were created by this pattern are highlighted with dashed lines, whereas the newly assigned symbolic names are in italics. Note that the nodes `$begin:s` and `$end:s` will be properly linked when the instrumenter processes the body statement matched by :s. This strategy makes the instrumentation description quite modular, since only one construction is taken care of at a time.

Figure 4.3 presents the instrumentation description of the *If-Then-Else* statement. Note that the same schema is applied. Lines 9 to 12 link the `$begin` node of the matched statement with the `$begin` nodes of its sub-statements and the `$end` nodes of the sub-statements to its `$end` node. Figure 4.4 presents the corresponding graph alteration for the *If-Then-Else* pattern. Note that this pattern also makes the same assumptions referred to in the *While* pattern, i.e. the nodes `$begin` and `$end` are already assigned for

```
1  pattern IfThenElse
2  var
3      :e  as [E]
4      :s1 as [S]
5      :s2 as [S]
6  match
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 checkpoint $begin:s1 before :s1
17     add checkpoint $begin:s2 before :s2
18     add checkpoint $end      after  self
19 end pattern
```

Figure 4.3: Instrumenting an *If-Then-Else* Statement.

every statement, and the only task of a pattern (w.r.t. nodes and edges) is to properly link the corresponding nodes for the matched statements, delegating the linkage of the nodes of their sub-statements to a respective pattern.

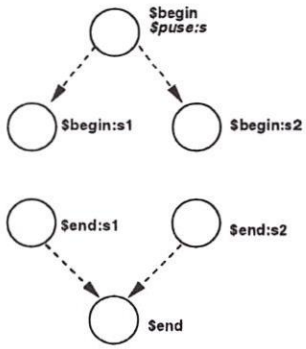



Figure 4.4: Graph Alteration during the Instrumentation of an *If-Then-Else* statement.

Figure 4.5 presents the instrumentation description of the *Break* statement. The only thing that should be done is to link the `$begin` node to the `$break` node (line 5). The semantics of the bottom-up search of symbolic names (discussed in 4.1.1) ensures that the innermost tree node that assigns a graph node to `$break` will be queried. Therefore, the instrumenter will respect the context of the *Break* command.

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

Figure 4.5: Instrumenting an *Break* Statement.

Figure 4.6 shows other two examples of pattern declarations that illustrate some interesting aspects of IDeL. The pattern declaration in lines 1 to 11 handles the assignment statement of the language. The control flows directly from the beginning to the end of the statement (line 8). Line 9 marks a definition of the identifier unified to :d at the graph node `$begin`. Line 10 assigns the symbolic name `$cuse` (that has the semantics of standing for computational use). Note that different treatment is employed to definitions and c-uses in the *Assignment* pattern. Whilst we actually mark a definition of :d, we postpone marking of the c-use in :e for another pattern by assigning the proper symbolic node. The reason for this difference is that :d stands for the actual identifier, whereas :e is an expression, composed of identifiers and other elements (e.g. operators). Thus, replacing Line 10 by

```
11'   mark cuse of :e at $begin
```

would mark a c-use of the whole expression, which is not the desired behavior.

The pattern declaration in lines 13 to 21 handles the uses of variables (identifiers, in this case). This declaration demands that, whenever an identifier is found, the instrumenter should mark a p-use and a c-use of the identifier in the nodes `$puse` and `$cuse`, respectively (lines 19 and 20). Recall that if a declaration refers to a symbolic name that is not assigned, this declaration is silently ignored. Therefore, the p- or c-use will be only marked if the respective symbolic name is actually defined. With this, one can control the kind of use of an identifier by assigning one or another of these symbolic names. Note that the *mark* declarations collect information about the program in the form of a relation that holds between an element of the program and a node in the graph, e.g. the meta-variable :d and the node `$begin` in line 9 and the meta-variable :u and the node `$puse` and `$cuse` in lines 20 and 21, respectively. Actually, this is the only kind of information IDeL currently supports. Note also that we mark p-uses not in the edges of control flow branches (as discussed in Section 2.1), but rather in the nodes where the predicate expressions are evaluated. Nevertheless, this does not limit the power of IDeL

```
1  pattern Assignment
2  var
3      :d as [ID]
4      :e as [E]
5  match
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 pattern
13
14 pattern Use
15 var
16     :u as [ID]
17 match
18     [ID< :u >]
19 graph
20     mark puse of :u at $puse:u
21     mark cuse of :u at $cuse:u
22 end pattern
```

Figure 4.6: Definition and Use Handling

in this case, since we can retrieve the edges a p-use should be used in by taking the node where the p-use was marked and checking what edges begin in it.

Figure 4.7 presents the program graph obtained after the application of the proper patterns to the program in Figure 3.3(a), shown in Section 3.1. By carefully analyzing the graph, one can easily identify which pattern was used to created each edge (every pattern described in this technical report was applied at least once in order to derive this graph). For example, the nodes `a`, `i`, `c`, `d` and `b` correspond, respectively, to the nodes `$begin`, `$control`, `$begin:s`, `$end:s` and `$end` in Figure 4.2. Note that `c` is the node `$begin:s` of the *While* pattern as well as the node `$begin` of the *If-Then-Else* pattern.

#### 4.1.2.3 Implementation Part

In the unit processing part, the *instrument* declarations are abstract. The actual instrumentation may vary from language to language. The implementation part gives a concrete syntax to the *instrument* declaration defined in the unit processing part and declares how to apply them to the language in hand. It is composed by a list of

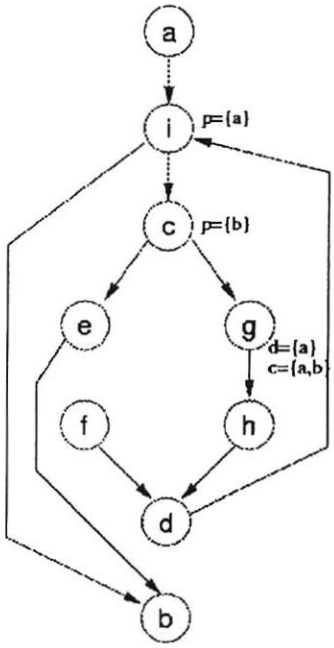



Figure 4.7: The Program Graph (before the reduction and rearrangement).

*implement* declarations. For example, the *instrument* declaration in line 18 of Figure 4.1 (a "checkpoint" before an expression) is implemented as shown in Figure 4.8.

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

Figure 4.8: Implementing a Checkpoint before an Expression

Lines 5 and 6 compose the pattern that should be matched for this *implement* declaration to be applied. Note that not only the tree pattern in line 6, but also the implement type and position in line 5 should match. So, this *implement* declaration will be applied only to a tree node that is an expression and has a checkpoint before itself. Line 9 is the pattern tree that will replace the matched tree node. Note that the checkpoint statement check(:n) is included before the expression matched by :e, separated by a comma. Assuming the usual C language semantics, this informally means that the

checkpoint statement will be executed before the expression is evaluated and will not affect the resulting value, which is the desired behavior. The meta-variable :n is bound to the graph node assigned to `$node` in order to be used in the substitution pattern[1].

### 4.1.3 Execution

The execution of an IDeL instrumentation description is ruled by three elements: a program $P$ to be instrumented (written in a language $L$), a context free grammar 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. the language $L$, its syntax tree is built.

2. **Unit Identification:** The syntax tree is traversed, starting from the root node, trying to match the patterns in the unit identification part of the instrumenter. Whenever a node in the syntax tree matches one of these patterns, a new graph is created and the matched node is processed (phase 3). If no node matches the unit patterns, the implementation phase begins (phase 4)

3. **Unit Processing:** IDeL executes every step of the instrumenter, in sequence. Each step is processed either from the root tree node down to leaf nodes or the other way around. (The direction of processing is defined when the step is declared.) For each tree node, each pattern declaration in the step is tried until one of them matches, if any. This pattern is then applied. After the execution of the last step, we employ a reduction algorithm to minimize the number of nodes, rearrange the nodes, and output the graph.

   (a) **Graph Reduction:** The program graph produced by IDeL may not be minimal w.r.t. the criterion of sequential execution presented in Section 3.1. For example, in Figure 4.7 the node `h` will always be executed after the node `g`. To minimize the graph, IDeL applies an algorithm 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. This is the case of node `f`.

[1]Note that `$node` is not part of the target language grammar and so, it cannot appear in a pattern. The binding allows the associations of meta variables to graph nodes.

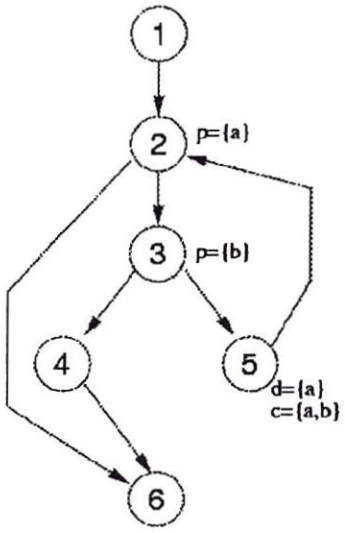



Figure 4.9: Reduced Program Graph of Graph in Figure 4.7.

(b) **Graph Rearrangement:** The graph rearrangement assigns labels to the nodes trying to resemble the program structure. This is often necessary after merging nodes by reduction. Figure 4.9 shows the program graph in Figure 4.7 after the reduction and rearrangement.

(c) **Graph Output:** The program graph is ready to be output. Currently, the program graph (its nodes, edges and the information marked in the nodes) is output in either XML or DOT (Gansner and North, 2001) format. The graph can be graphically displayed by post-processing it with graph drawing application, e.g. GraphViz (Gansner and North, 2001) and GraphView (Vilela et al., 1997).

After outputing the graph, the instrumenter resumes in phase 2, looking for another unit.

4. **Implementation:** The syntax tree is traversed, starting from the root node, checking in the implementation tables whether there is an implementation to be made. When an implementation is requested, IDeL searches in the *implement* declarations for one that is applicable for the respective node, type and position. If an applicable *implement* declaration is found, the syntax tree is accordingly changed and the tree traversal goes on.

5. **Unparsing:** The syntax tree is traversed and every terminal is collected. The sequence of terminals so obtained is the instrumented program.

## 4.2 IDeLgen: Operational Aspects

Given a grammar, a program in that grammar, and an instrumentation description for that grammar, we should be able to generate the respective program graphs and the instrumented program. For this task, we developed the IDeLgen (standing for IDeL Generator). Its overall execution schema is shown in Figure 4.10. When IDeLgen is input with a grammar, say grm, it produces a program called IDeL.grm. In its turn, this program is to be run with an instrumentation description *desc* and a program $P$.

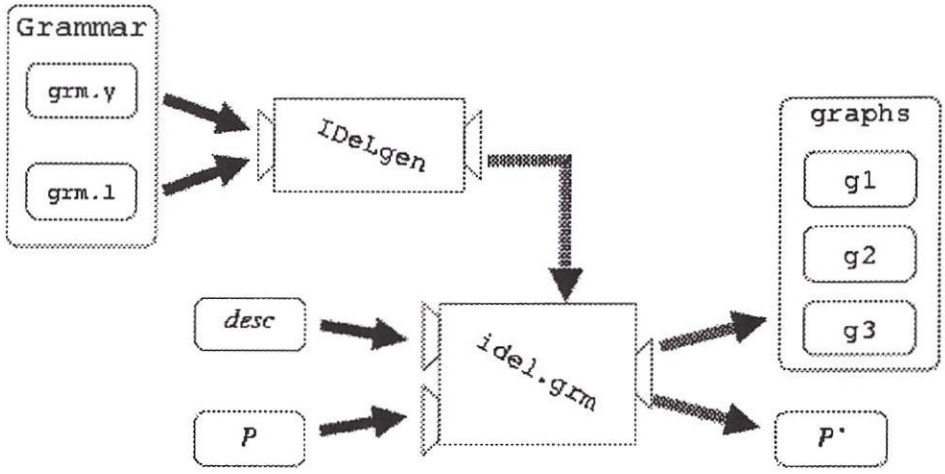



Figure 4.10: IDeLgen Execution Schema

The grammar input to IDeLgen is furnished in two files. One of these files is a lexical analyzer input in a .l file, written in a subset of the lex syntax (Mason and Brown, 1990). The other file is the context-free grammar input in a .y file, written in a subset of the yacc syntax (Mason and Brown, 1990). Indeed, these files can be thought of as minimal standard yacc and lex inputs, from which all so-called semantic actions were removed.

An example of a typical execution with IDeLgen, considering C language, is presented bellow.

```
hortencia% IDeLgen C
hortencia% IDeL.C C.idel Sample.C
```

In the first line, IDeLgen is called with the option C, which means that files named C.l and C.y are to be read. Then, the program IDeL.C is produced. In the second line, this program is run with the files C.idel and Sample.C, which are the instrumentation description and the original program, respectively. As a result, both an instrumented program and a set of program graphs are generated.

## 4.3 Some Empirical Results

Using the language proposed in this technical report, we were able to describe an instrumenter for C language based on the instrumentation schema proposed by Maldonado in (Maldonado, 1991). We developed this instrumenter as part of a prototype tool for supporting analyses of control and data flow based testing criteria. (In this tool, the analyses are actually accomplished by Prolog predicates, in the same vein as discussed in (Simão et al., 2001).)

Starting from the instrumentation description for C language, we developed instrumenters for C++ and Java languages. We were able to reuse most parts of the C instrumenter description, since both C++ and Java are syntactically very similar to C and their structures resemble one another very closely. The part of the prototype responsible for the analysis remains unchanged. This experience indicates that IDeL can be used in a multi-language approach, taking advantage of language similarities to reuse the instrumentation description.

We have also used IDeL to instrument C programs in order to collect the information required for a strategy that employs genetic algorithms to derive test cases. We based the instrumenter in the informal description provided in (Bueno and Jino, 2001). Here again, starting from the instrumentation description for testing criteria analyses, we had only to alter a few patterns and change the implement section accordingly. This experience indicates that IDeL can also be used in different contexts.

Chapter

5

# Concluding Remarks

Program instrumentation is a technique largely employed in software engineering, suitable to obtain an abstract view of the program and to get more insight about the program execution. This instrumentation is usually conducted with a specific purpose in mind, such as testing coverage assessment and program visualization.

In this technical report we presented a language for describing instrumentation. The 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 some important data about the program and insert log statements to register the program execution.

Although IDeL can be used in several distinct contexts and applications, some of the design decisions it embodies were indeed biased by our main objective (i.e. being able to instrument programs for supporting testing criteria coverage analysis). As a consequence, some constructions are closely related to nodes and edges. For example, IDeL provides only one declaration for marking information relating one element of the program with one node of the graph. However, the language should be generalized in order to allow for other kinds of information to be handled, e.g. edges, constant values, more than one node or program element etc. A future step in this research is to identify what kinds of other information should also be collected and include new declarations to handle them.

Currently, IDeL does not provide mechanisms to cope with *goto* statements. Actually, a *goto* statement demands that nodes be referred through 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 of some element of the program and we are studying how to generalize the construction to be useful for the more general cases, such as referring to entry and return nodes of an inter-procedural call (Harrold and Soffa, 1991).

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. 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 instrumentation description requires that just the relevant parts of the grammar be considered and, by carefully constructing the grammar (choosing nonterminal names consistently) and the description (isolating features particular to a language) even the description can be reused or, at least, part of it.

# Bibliography


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

Bratko, I. (1990). *Prolog Programming for Artificial Intelligence.* Addison-Wesley, 2 edition.

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.

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.

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.

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.

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.

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

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.

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

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

Neighbors, J. (1984). The draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5).

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

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

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.

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.

Vilela, P. R., Maldonado, J. C., and Jino, M. (1997). Program graph visualization. *Software-Practice & Experience*, 27(11):1245–1262.

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

Appendix

A

# IDeL Grammar

This appendix presents the grammar of an IDeL description file.

⟨*InstrInstrumenter*⟩ ::= '`instrumenter`' ⟨*ID*⟩ ⟨*InstrUnitDeclSct*⟩ ⟨*InstrSteps*⟩
⟨*InstrImplementationDeclSct*⟩ '`end`' '`instrumenter`'

⟨*InstrUnitDeclSct*⟩ ::= ⟨*InstrUnitDecls*⟩

⟨*InstrUnitDecls*⟩ ::= ⟨*InstrUnitDecl*⟩

⟨*InstrUnitDecl*⟩ ::= '`unit`' ⟨*InstrVarDeclSct*⟩ ⟨*InstrNamedBySct*⟩ ⟨*InstrPatternList*⟩
'`end`' '`unit`'

⟨*InstrNamedBySct*⟩ ::= $\epsilon$
| '`named`' '`by`' ⟨*VARIABLE*⟩

⟨*InstrPatternList*⟩ ::= '`match`' ⟨*Pattern*⟩

⟨*InstrSteps*⟩ ::= ⟨*InstrStep*⟩
| ⟨*InstrSteps*⟩ ⟨*InstrStep*⟩

⟨*InstrStep*⟩ ::= '`step`' ⟨*ID*⟩ ⟨*InstrOrientation*⟩

⟨*InstrOrientation*⟩ ::= $\epsilon$
| '`TB`'
| '`BT`'

⟨*InstrPatternDecls*⟩ ::= ⟨*InstrPatternDecl*⟩
| ⟨*InstrPatternDecls*⟩ ⟨*InstrPatternDecl*⟩

⟨*InstrPatternDecl*⟩ ::= '`pattern`' ⟨*ID*⟩ ⟨*InstrVarDeclSct*⟩ ⟨*InstrMatchDeclSct*⟩
⟨*InstrNodeDeclSct*⟩ ⟨*InstrGraphDeclSct*⟩ ⟨*InstrAssignmentDeclSct*⟩
⟨*InstrInstrumentDeclSct*⟩ '`end`' '`pattern`'

⟨*InstrVarDeclSct*⟩ ::= $\epsilon$
| '`var`' ⟨*InstrVars*⟩

⟨*InstrVars*⟩ ::= ⟨*InstrVar*⟩
| ⟨*InstrVars*⟩ ⟨*InstrVar*⟩

⟨*InstrVar*⟩ ::= ⟨*VARIABLE*⟩ '`as`' ⟨*InstrVariableType*⟩

⟨*InstrMatchDeclSct*⟩ ::= '`match`' ⟨*Pattern*⟩

⟨*InstrNodeDeclSct*⟩ ::= ⟨*InstrNodeDecls*⟩

⟨*InstrNodeDecls*⟩ ::= $\epsilon$
| ⟨*InstrNodeDecls*⟩ ⟨*InstrNodeDecl*⟩

⟨*InstrNodeDecl*⟩ ::= '`declare`' '`node`' ⟨*NODE*⟩

⟨*InstrGraphDeclSct*⟩ ::= $\epsilon$
| '`graph`' ⟨*InstrGraphDecls*⟩

⟨*InstrGraphDecls*⟩ ::= ⟨*InstrGraphDecl*⟩
| ⟨*InstrGraphDecls*⟩ ⟨*InstrGraphDecl*⟩

⟨*InstrGraphDecl*⟩ ::= ⟨*InstrLinkDecl*⟩
| ⟨*InstrDefinitionDecl*⟩

⟨*InstrLinkDecl*⟩ ::= ⟨*InstrNodeReference*⟩ '->' ⟨*InstrNodeReference*⟩

⟨*InstrNodeReference*⟩ ::= ⟨*NODE*⟩
| ⟨*NODE*⟩ ⟨*VARIABLE*⟩

⟨*InstrDefinitionDecl*⟩ ::= 'mark' ⟨*ID*⟩ 'of' ⟨*VARIABLE*⟩ 'at' ⟨*InstrNodeReference*⟩

⟨*InstrAssignmentDeclSct*⟩ ::= $\epsilon$
| 'assignment' ⟨*InstrAssignDecls*⟩

⟨*InstrAssignDecls*⟩ ::= ⟨*InstrAssignDecl*⟩
| ⟨*InstrAssignDecls*⟩ ⟨*InstrAssignDecl*⟩

⟨*InstrAssignDecl*⟩ ::= 'assign' ⟨*InstrNodeReference*⟩ 'to' ⟨*InstrNodeReference*⟩

⟨*InstrInstrumentDeclSct*⟩ ::= $\epsilon$
| 'instrument' ⟨*InstrInstrumentDecls*⟩

⟨*InstrInstrumentDecls*⟩ ::= ⟨*InstrInstrumentDecl*⟩
| ⟨*InstrInstrumentDecls*⟩ ⟨*InstrInstrumentDecl*⟩

⟨*InstrInstrumentDecl*⟩ ::= 'add' ⟨*ID*⟩ ⟨*InstrNodeReference*⟩ ⟨*InstrPosition*⟩
⟨*InstrVariableOrSelf*⟩

⟨*InstrPosition*⟩ ::= 'before'
| 'after'

⟨*InstrVariableOrSelf*⟩ ::= ⟨*VARIABLE*⟩
| 'self'

⟨*InstrImplementationDeclSct*⟩ ::= $\epsilon$
| 'implementation' ⟨*InstrImplementDecls*⟩

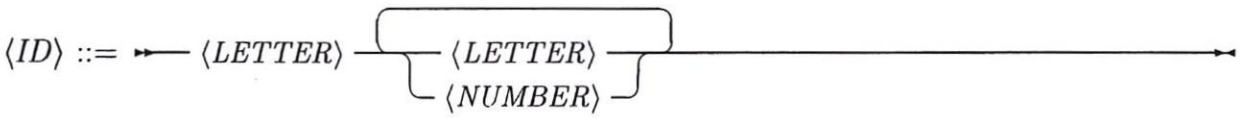
⟨*InstrImplementDecls*⟩ ::= ⟨*InstrImplementDecl*⟩
| ⟨*InstrImplementDecls*⟩ ⟨*InstrImplementDecl*⟩

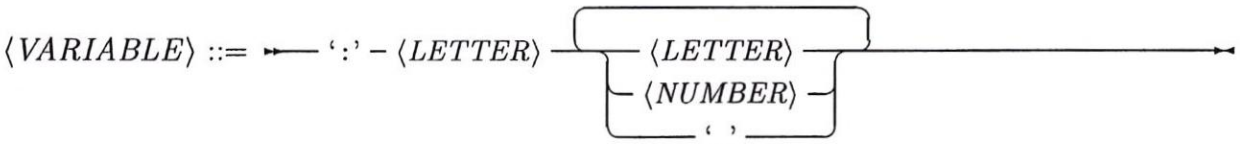
⟨*InstrImplementDecl*⟩ ::= 'implement' ⟨*InstrVarDeclSct*⟩ ⟨*ID*⟩ ⟨*NODE*⟩ ⟨*InstrPosition*⟩
⟨*Pattern*⟩ ⟨*InstrBindingSct*⟩ 'as' ⟨*Pattern*⟩ 'end' 'implement'

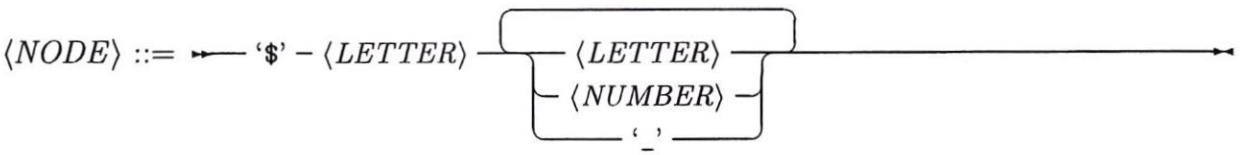
⟨*InstrBindingSct*⟩ ::= ⟨*InstrBindingDecls*⟩

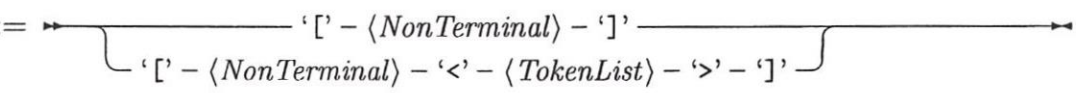
⟨*InstrBindingDecls*⟩ ::= $\epsilon$
| ⟨*InstrBindingDecls*⟩ ⟨*InstrBindingDecl*⟩

⟨*InstrBindingDecl*⟩ ::= 'binding' ⟨*VARIABLE*⟩ 'to' 'node' ⟨*NODE*⟩
| 'binding' ⟨*VARIABLE*⟩ 'to' 'literal' ⟨*Pattern*⟩



⟨*Pattern*⟩ ::= 


**Note:** ⟨*NonTerminal*⟩ is a non-terminal symbol of the target grammar and ⟨*TokenList*⟩ is a list of tokens (and meta-variables therein) that can be derived from ⟨*NonTerminal*⟩ in that grammar.

⟨*LETTER*⟩ ::= 'a' .. 'z', 'A' .. 'Z'

⟨*NUMBER*⟩ ::= '0' .. '9'