Testing Attribute Grammars

Jörg Harm[1] and Ralf Lämmel[2]

1: Universität Rostock, Fachbereich Informatik, D-18051 Rostock
jh@informatik.uni-rostock.de
2: CWI, P.O. Box 94079, NL-1090 GB Amsterdam
Ralf.Laemmel@cwi.nl

Abstract

Fundamental notions for testing attribute grammars are developed. Two dimensions are explored. The structural dimension focuses on the context-free grammar part of an attribute grammar, whereas the semantic dimension is concerned with attributes, attribute types, conditions, and computations. In both dimensions, and also for the combination of them, we are interested in coverage notions, test set criteria (e.g., minimum test sets), and test set generation. The described framework builds upon abstract interpretation technology, and search algorithms. The approach is also applicable to some other representatives of the declarative paradigm, such as logic programming and term rewriting. Examples illustrate that the framework is particularly useful in testing language definitions based on attribute grammar specifications.

1. Introduction

Technology for testing declarative programs is less established than for mainstream programming. This is particularly true for testing attribute grammar specifications—the topic of this paper. A major application area for attribute grammars (AGs) is language definition, prototyping and implementation. Developing, extending and tuning real-world AG specifications are non-trivial tasks. Thus, testing and verification should form a standard activity in the corresponding software engineering processes. Verification has been addressed in the literature to a certain extent (cf. [CD84b, LR94]). On the contrary, testing is poorly developed for AGs. There is, for example, no standard pool of testing techniques for AGs available. There do not even exist sensible non-trivial coverage notions for test sets on AGs. Standard testing technology (cf. [Mye79, Bei90]) is not applicable to a large extent.

In this paper, we present a complete framework for testing AG specifications with a slight emphasis on language definition and implementation. Two dimensions of testing AGs are explored. The underlying context-free grammar of an AG induces the *structural* dimension, whereas attributes, attribute types, computations and conditions induce the *semantic* dimension. The structural dimension has been explored to a certain extent by previous literature. The coverage notions for test sets go hardly beyond rule coverage (cf. [Pur72]). Existing approaches are usually also not intuitive in the sense that they would be concerned with aspects of the language. They are also not flexible enough in that they just induce certain test sets for the entire grammar, but it is hard to focus on a certain aspect as, for example, the number of recursive unfoldings of a certain nonterminal. The semantic dimension and its combination with the structural dimension has not been investigated except a few attempts to generate correct derivation trees (cf. [Kas80, HLR97, Har97]), i.e., trees which can be decorated and the conditions on the attributes are satisfied.

The major technical contribution of this paper is a notion of coverage addressing the two dimensions in separation and in combination. The notion is general in the sense that it goes very much beyond just rule coverage. It also can cope with the recursion involved in the underlying grammar and in the attribute type definitions. The coverage notion is effective in the sense that it can be used for test set generation for AG specifications. The generation might be started from the pure specifications. Additional information can be provided to put the focus on a certain aspect or to enforce proper termination in certain situations.

Opportunities for the application In this paper, we mainly emphasize the common point of view that testing is useful to gain confidence about the correctness of the tested specification or implementation. By generating and applying test cases for a language definition specified in terms of an AG, the developer can check if his intuitions regarding the language are met, and if the behaviour of the executable language definition is as expected. This process can be conceived as a kind of white-box debugging of language definitions.

There are the following other opportunities in the context of language definition and implementation. The developed coverage criterion is useful in assessing given test sets or existing test suites (cf. [BS94]). Test sets generated from an AG specification are also useful for a kind of black-box testing of language implementations where a given AG specification is considered as the reference. In that case, the actual implementation does not need to be based on AGs at all, or if it is (partially) based on compiler compiler (say AG) technology, the actual specification used for the implementation might not be accessible, or it might deviate from the reference specification for practial reasons.

Structure of the paper In Section 2, some basic notions needed in the main part of the paper are spelled out. The reader might skip this section for a first reading. In Section 3, our testing approach is motivated. An AG describing syntax and (part of the) static semantics of a language with blocks and jumps serves as an example. In Section 4, the new notion of coverage is developed. In Section 5, test set generation is discussed. The goal is to generate test sets achieving full coverage. In this context, termination and search space problems need to be addressed. In Section 6, the paper is concluded.

2. Basic notions

A minimum amount of basic notions for dealing with attribute grammars is provided. The reader might skip this section for a first reading, and go directly to Section 3 motivating our approach. We assume basic knowledge of context-free grammar theory and attribute grammars as covered by surveys like [Sud88, Alb91, Paa95].

We adopt the common algebraic interpretation of context-free grammars and the standard Dewey notation for decorated derivation trees, including partial ones. We also explain how partial trees can be extended in the sense of derivation. These terms are useful in test set generation, where complete trees are derived in a stepwise manner. We need an interpretation for the attribute types and the relation and function symbols used to formulate conditions and computations. By this, we can decide for a complete tree, if it encodes a proper word generated by the underlying AG. The interpretation might also be used for intermediate partial trees during test set generation in order to reject unsatisfiable trees as early as possible.

2.1. Context-free and attribute grammars

A context-free grammar G is a quadruplet $\langle N, T, s, P\rangle$ as usual, i.e., N and T are the disjoint finite sets of nonterminals resp. terminals. $s \in N$ is called start symbol. P is a finite set of productions or (context-free) rules with $P \subset N \times (N \cup T)^{\star}$. We resort to the common notation $l \rightarrow r$ for a production $p = \langle l, r\rangle \in P$ with $l \in N$ and $r \in (N \cup T)^{\star}$. For simplicity, we assume reduced context-free grammars in the sequel.

An attribute grammar AG is a quadruplet $\langle G, A, CM, CN \rangle$, where G is the underlying context-free grammar, A associates each $x \in N \cup T$ with finite sets of synthesized attributes $A_s(x)$ and inherited attributes $A_i(x)$, CM and CN associate each production p of G with finite sets of computations $CM(p)$ and conditions $CN(p)$. We assume well-formed, non-cyclic attribute grammars in normal form. Given a production $p = x_0 \to x_1 \cdots x_m \in P$, with $x_0 \in N$, $x_1, \ldots, x_m \in N \cup T$, a computation c from $CM(p)$ is of the form $r_0.a_0 := f_c(r_1.a_1, \ldots, r_k.a_k)$ where $0 \leq r_j \leq m$, and x_{r_j} carries an attribute a_j for $j = 0, \ldots, k$. The notation for conditions is similar.

An example attribute grammar will be provided in Section 3.

2.2. Complete and partial derivation trees

To reason about derivation trees, a variant of the common algebraic interpretation of context-free grammars is used. Given a context-free grammar $G = \langle N, T, s, P \rangle$, a signature Σ is derived as follows. $N \cup T$ provide the sorts of Σ. We need to include T because we want to represent terminals in the decorated derivation trees, since they may carry (synthesized) attributes. The productions in P are considered as function symbols in Σ, i.e., given a $p = x_0 \to x_1 \cdots x_m \in P$, there is a function symbol $p : x_1 \times \cdots \times x_m \to x_0 \in \Sigma$. Since we included terminals in Σ, we need to declare a term representation for terminals. We assume a special constant symbol $\mathit{leaf}_x : \to x$ for each $x \in T$ in Σ. Without further effort, *complete* derivation trees can be regarded as terms of the term algebra $T_s(\Sigma)$. The representation of *partial* derivation trees is not so straightforward. One option to cope with "holes" in derivation trees is to consider terms with variables, i.e., a partial derivation tree derived from a nonterminal n is a term from $T_n(\Sigma, X)$ over some N-sorted family of variables X. This option is not quite usable. Variables are rather non-intuitive representations of holes in a partial derivation tree, since there is no natural interpretation for multiple occurrences of one variable in a context-free derivation tree. We resort to another option, that is we assume a constant symbol $\mathit{leaf}_x : \to x$ for each $x \in N$ in Σ. Recall that for terminals, leaf_x denotes a leaf in the sense of a complete derivation tree, whereas for nonterminals, leaf_x denotes a hole in the derivation tree. Derivation steps are meant to replace such holes.

Derivation trees might be obtained essentially in two ways. First, a rule can be represented as a partial derivation tree. Second, partial derivation trees can be extended by replacing holes by further derivation trees. We will present these two fundamental concepts. Afterwards, the decoration of derivation trees is addressed. Note that for technical reasons, we also assume that leaf_x for each $x \in N$ is a proper partial derivation tree.

Given a production $p = x_0 \to x_1 \cdots x_m \in P$, the corresponding partial derivation tree is represented by the term $p(\mathit{leaf}_{x_1}, \ldots, \mathit{leaf}_{x_m})$. We need to introduce a notation for selecting or addressing subterms in the spirit of VDL (cf. [Weg72]). Given a term $t \in T_n(\Sigma)$ and a selector sequence $q^\star \in \mathcal{N}_0^\star$, selection is defined as follows:

$$\mathcal{S}(t, q^\star) = \begin{cases} t, & \text{if } q^\star = \langle\rangle \\ \mathcal{S}(t_i, \langle q_2, \ldots, q_l \rangle), & \text{if } t \text{ is of the form } p(t_1, \ldots, t_k), \\ & q^\star = \langle q_1, \ldots, q_l \rangle, q_1 = i \in \{1, \ldots, k\} \\ \text{undefined}, & \text{otherwise} \end{cases}$$

In a similar way, the extension of t by $t' \in T_{n'}(\Sigma)$ at the leaf addressed by $q^\star$ can be defined:

$$\mathcal{E}(t, q^\star, t') = \begin{cases} t', & \text{if } t = \mathit{leaf}_{n'}, q^\star = \langle\rangle \\ p(t_1, \ldots, t_{i-1}, t'_i, t_{i+1}, \ldots, t_k), & \text{if } t \text{ is of the form } p(t_1, \ldots, t_k), \\ & q^\star = \langle q_1, \ldots, q_l \rangle, \\ & q_1 = i \in \{1, \ldots, k\}, \\ & t'_i = \mathcal{E}(t_i, \langle q_2, \ldots, q_l \rangle, t') \\ \text{undefined}, & \text{otherwise} \end{cases}$$

2.3. Decorated derivation trees

It is easy to cope with *decorated* derivation trees. Essentially, we need to decide how to represent conditions and computations associated with the attributes of the nodes in a (pure) derivation tree. Using the standard Dewey notation as for selection, attributes are made unique within a tree. This is a standard technique outlined here for convenience. We regard a decorated derivation tree with root symbol n to be an element of $\mathcal{T} = T_n(\Sigma) \times \mathcal{P}(\mathcal{CM}) \times \mathcal{P}(\mathcal{CN})$, where $\mathcal{CM}$ and $\mathcal{CN}$ model computations resp. conditions of the decorated derivation tree. Thus, in contrast to CM and CN, Dewey sequences are used in $\mathcal{CM}$ and $\mathcal{CN}$ to refer to the nodes of attributes.

Let us describe how the partial decorated derivation tree $\langle t, cm, cn\rangle$ corresponding to a single rule $p = x_0 \to x_1 \cdots x_m \in P$ is obtained. We already described that the context-free part t is $p(\mathit{leaf}_{x_1}, \ldots, \mathit{leaf}_{x_m})$. From a computation $r_0.a_0 := f(r_1.a_1, \ldots, r_k.a_k) \in CM(p)$ with the r_i referencing grammar symbols in p, the following contribution to cm is derived:

$$
\begin{array}{rcl}
q_0^\star.a_0 & := & f(q_1^\star.a_1, \ldots, q_k^\star.a_k), \text{where} \\
& & q_i^\star = \left\{ \begin{array}{ll} \langle\rangle & , \text{if } r_i = 0 \\ \langle r_i \rangle & , \text{if } 1 \leq r_i \leq m \end{array} \right. \\
& & \text{for } i = 0, \ldots, k
\end{array}
$$

Similarly, the conditions $CN(p)$ are turned into cn. Thus, a partial decorated derivation tree corresponding to a production consists of the partial derivation tree + the computations and conditions translated to their encodings in $\mathcal{CM}$ and $\mathcal{CN}$. Extension of a tree t by another tree t' at $q^\star$ is a straightforward elaboration of the context-free case. The selector sequences used in the computations and conditions of t' need to be prefixed by $q^\star$. Extension on decorated trees is defined if the context-free extension succeeds.

It is clear that the computations in a decorated tree t induce an equational system whose solution—if it exists—supplies the attribute values for t. Note that the solution is also constrained by the conditions of t.

3. Motivation by example

What does it mean to test an attribute grammar thoroughly? To answer this question, we can investigate the two parts of an attribute grammar, the context-free base grammar, and the attributes with conditions and computations, separately and in combination. For the moment being, we restrict ourselves to positive test cases.

To test an acceptor A which is supposed to accept a context-free language L, we take some finite set $T \subseteq L$ and ask if A accepts each $w \in T$. We want to gain confidence that the language accepted by A actually is L. Thus, we have to ensure that T reflects to a certain degree all aspects of L. Since a context-free grammar G for L necessarily reflects all aspects of L, it suffices to cover G by T in some sense. The bare minimum of coverage is to claim that every production of G is applied in the derivation of some $w \in T$. Actually, this coverage notion is by far too simple.

Figure 1 shows an excerpt of a context-free grammar for a Pascal-like programming language.

In the derivation of the program

`label a; begin a : goto a; begin if ... then ... end end.`

all productions of the context-free grammar of Figure 1 are used. More complex criteria than simple rule coverage are sensible to enforce certain kinds of combinations of productions. Looking, for

[prog]	Prog	$\rightarrow$	Block .
[block]	Block	$\rightarrow$	Decls `begin` Stms `end`
[nodecl]	Decls	$\rightarrow$	ϵ
[decls]	Decls	$\rightarrow$	Decl Decls
[decl]	Decl	$\rightarrow$	`label` id ;
[onestm]	Stms	$\rightarrow$	Stm
[stms]	Stms	$\rightarrow$	Stm ; Stms
[goto]	Stm	$\rightarrow$	`goto` id
[ldef]	Stm	$\rightarrow$	id : Stm
[if]	Stm	$\rightarrow$	`if` ... `then` Stm
[localb]	Stm	$\rightarrow$	Block
...			

Figure 1: Part of a context-free grammar for a Pascal-like language

example, at the declaration part of a block, the following aspects were not reflected in the program above:

1. The declaration part may consist not only of one, but also of zero or more than one declaration.
2. A block used in place of a statement may have a non-empty declaration part.

Therefore, we need a coverage notion which treats recursion in a sensible way (1.), and which is more context-dependent than just rule coverage (2.).

The idea presented here follows a bottom-up approach. Starting from a trivial coverage notion for the terminals of the grammar, we build a layered coverage notion for the nonterminals taking a predefined number of recursive unfoldings into account. For terminals like `begin`, coverage means just to use them, for terminals which actually correspond to a terminal class, e.g., id, coverage may be to use one or two different representatives. As a first attempt, for a nonterminal A with A $\rightarrow w$ as one of its alternatives, we can say that this alternative is covered if all grammar symbols in w are covered. A nonterminal is covered if all alternatives for this nonterminal are covered. Of course, this definition of coverage has to be refined to cope with recursively defined nonterminals in a sensible way. Coverage is redefined w.r.t. a counter η mapping nonterminals to numbers of recursive unfoldings. Now, we say that A $\rightarrow w$ is covered w.r.t. η if all grammar symbols in w are covered w.r.t. η decremented for A.

Two recursive unfoldings should suffice in most cases. Indeed, in our example (see 1. above), a test set with programs with zero, one and more than one declarations in the block of a program and in the block of a statement would achieve coverage. Note that context-dependency of coverage (as needed in 2. above) is achieved by defining coverage for all rules in separation, namely by decrementing η in a context.

Now, we can take a look at the attribute part of an attribute grammar. For our Pascal-like programming language, the attribute part is concerned with static semantics of the language. In the attribute grammar fragment in Figure 2, we focus on scope rules for labels. We use lists of identifiers, that is the domain $\mathcal{ID\_LIST}$, for the representation of sets of labels.

The coverage notion for context-free grammars can be lifted to attribute grammars if we require that the base grammar is covered with a subset of the language defined by the attribute grammar. In addition, we can go one step further by taking the structure of the attribute values into account. The idea of a layered coverage notion spelled out for context-free grammars above can be adapted to attribute type definitions in a sensible way. Thereby, we get a coverage notion for possibly recursive domain equations. Suppose that the basic domain $\mathcal{ID}$ is covered by two different representatives, we can cover the domain $\mathcal{ID\_LIST}$ in the case of recursion level 2 by lists of the length zero, one, and

Type definitions:

$$\mathcal{ID} = \{a, ..., z\}^+$$
$$\mathcal{ID\_LIST} = [\,] + [\mathcal{ID} | \mathcal{ID\_LIST}]$$

Attributes:

$A_i(\text{Block}) = A_i(\text{Stms}) = A_i(\text{Stm}) = \{\text{TL}\}$
$A_s(\text{Stms}) = A_s(\text{Stm}) = \{\text{DL}, \text{LTL}\}$
$A_s(\text{id}) = \{\text{Name}\}$
$A_s(\text{Decl}) = \{\text{LN}\}$
$A_s(\text{Decls}) = \{\text{L}\}$

Attribute types:

Name **:** $\mathcal{ID}$	name of the identifier
LN **:** $\mathcal{ID}$	name of the declared label
L **:** $\mathcal{ID\_LIST}$	list of declared labels
TL **:** $\mathcal{ID\_LIST}$	target labels reachable from inside the block, statement list, or statement
DL **:** $\mathcal{ID\_LIST}$	labels with a defining occurrence inside the statements of a block
LTL **:** $\mathcal{ID\_LIST}$	labels with defining occurence inside the statement list or statement which are reachable by goto statements on the same statement nesting level

[prog]	Prog	→	Block <u>.</u>	1.TL := [ ]
[block]	Block	→	Decls <u>`begin`</u> Stms <u>`end`</u>	3.TL := (0.TL \ 1.L) ∪ 3.LTL
				3.DL = 1.L
[nodecl]	Decls	→	ϵ	0.L := [ ]
[decls]	Decls	→	Decl Decls	0.L := [1.LN|2.L]
				1.LN ∉ 2.L
[decl]	Decl	→	<u>`label`</u> id <u>;</u>	0.LN := 2.Name
[onestm]	Stms	→	Stm	1.TL := 0.TL
				0.DL := 1.DL
				0.LTL := 1.LTL
[stms]	Stms	→	Stm <u>;</u> Stms	1.TL := 0.TL
				3.TL := 0.TL
				0.DL := 1.DL ∪ 3.DL
				0.LTL := 1.LTL ∪ 3.LTL
				1.DL ∩ 3.DL = [ ]
[goto]	Stm	→	<u>`goto`</u> id	0.DL := [ ]
				0.LTL := [ ]
				2.Name ∈ 0.TL
[ldef]	Stm	→	id <u>:</u> Stm	3.TL := 0.TL
				0.DL := [1.Name|3.DL]
				0.LTL := [1.Name|3.LTL]
				1.Name ∉ 3.DL
[if]	Stm	→	if ... <u>`then`</u> Stm	4.TL := 0.TL ∪ 4.LTL
				0.DL := 4.DL
				0.LTL := [ ]
[localb]	Stm	→	Block	1.TL := 0.TL
				0.DL := [ ]
				0.LTL := [ ]
...				

Figure 2: Part of an AG for a Pascal-like language

greater than one where at each position at least two different $\mathcal{ID}$ values occur, e.g., $[\,]$, $[\mathtt{a}|[\,]]$, $[\mathtt{a}|[\mathtt{b}|[\,]]]$, and $[\mathtt{b}|[\mathtt{a}|[\,]]]$. Another example of a set covering $\mathcal{ID\_LIST}$ is $\{[\,], [\mathtt{a}|[\,]], [\mathtt{a}|[\mathtt{a}|[\,]]], [\mathtt{b}|[\mathtt{b}|[\,]]]\}$. Now, we can say a subset T of the language defined by an attribute grammar AG covers an attribute of a production of AG, if the values associated with this attribute in the derivation trees of the elements of T cover the domain of the attribute. While the coverage notion for context-free grammars forces the application of grammar rules in meaningful syntactical contexts, the domain coverage forces meaningful semantic contexts. Applied to the domain equation of Figure 2, the domain coverage criterion enforces the use of the alternatives of the nonterminal Stm in different contexts covering the attributes TL, DL, and LTL.

For both the syntactical and the semantic dimension, full coverage often can not be achieved for the initial AG specification because of two related problems:

- Full coverage for the context-free grammar may not be achieved because the conditions on the attributes prohibit some syntactical combinations.
- Full coverage may not be achieved for some attributes because in every decorated derivation tree the attribute values is of a special form.

These problems are somewhat similar to the well-known problem of unexecutable pathes in testing imperative programs. An example for the second problem is the 0.TL attribute of production [ldef] in Figure 2 which is always a non-empty list because at least the label defined by the alternative itself will be in the list. There is usually a sensible way to relax coverage explicitly. We postpone with a more detailed discussion of this issue.

4. Attribute grammar coverage

Ultimately, we want to define a coverage notion for AGs. We will derive this notion from a more fundamental coverage notion for context-free grammars and attribute type. We mentioned already that coverage for context-free grammars and attribute types can be defined in a similar way. For that reason, we define the fundamental coverage notion in the formal setting of a certain kind of equational systems which are meant to cover

- context-free grammars (if we resort, e.g., to a BNF-like interpretation), and
- attribute type definitions (if we resort, e.g., to recursively defined domains).

First, we define equational systems. Then, an abstraction scheme for equational systems to model coverage is developed. Afterwards, the scheme is instantiated to obtain the desired notation of coverage for context-free grammars and attribute types. Finally, the ultimate notion of AG coverage is derived.

4.1. Equational systems

An equational system S over variables $X_1, \ldots, X_n$ and constants $C_1, \ldots, C_m$ consists of n equations $X_1 \equiv t_1, \ldots, X_n \equiv t_n$, where the t_i are terms over $\mathbf{1}$, $X_1, \ldots, X_n$, $C_1, \ldots, C_m$ composed with $\times$ and $+$. A set-theoretic interpretation is assumed, i.e., $C_1, \ldots, C_m$ are predefined sets, $\mathbf{1}$ is a dedicated singleton set, $\times$ resp. $+$ correspond to Cartesian product and disjoint union. We assume a well-formedness property for S in the sense of the termination property for context-free grammars. The solutions for the X_i are denoted by $[\![X_i]\!]$.

It should be clear that this kind of equational systems can be used to study context-free grammars and attribute types in a unified setting. For convenience, we list a few arguments to support this claim. Attribute types can directly be modelled with equational systems if they are defined via datatype constructors, products, and sums. If we assume instead (first-order + non-parameterized) algebraic datatypes, such a definition can easily be transformed into an equivalent equational system by encoding constructors as sums of products. Context-free languages can be defined in various ways, e.g., by using context-free grammars, or by resorting to BNF/EBNF notation, or to an algebraic interpretation. The relation between these formalisms or notations is well understood. The kind of equational systems we propose is semantically best conceived as BNF. The terminals of a context-free grammar correspond to the C_i, whereas the nonterminals are the variables X_i. A union in a BNF (alternatives in a context-free grammar) is represented with $+$, whereas $\times$ models concatenation. $\mathbf{1}$ models an empty concatenation, i.e., ϵ.

4.2. An abstraction scheme to model coverage

The $[\![X_i]\!]$ for an equational system S are potentially infinite sets. Thus, an exhaustive test exploring all elements of $[\![X_i]\!]$ is impractical. We need to perform abstraction to derive a feasible coverage notion. Abstract interpretation concepts are used in the sequel (cf. [Fos87, CM92, Ros92, JN95]).

The basic idea is that the X_i are associated with abstract domains $\overline{X_i}$ modelling coverage for X_i. The $\overline{X_i}$ are supposed to obey the following structure:

- They are posets of the form $\langle \overline{z}, \perp, \top, \leq \rangle$.
- $\overline{z}$ is a (not necessarily finite) set.
- $\leq$ is a partial order on $\overline{z}$.
- $\perp \in \overline{z}$ is the smallest element (modelling no coverage).
- $\top \in \overline{z}$ is the greatest element (modelling full coverage).
- For all elements a and b, the supremum $a \sqcup b$ is defined.
- There are only finite chains.

Another requirement will be added later. The finite chain property ensures that a finite test is sufficient to reach $\top$. In test set generation, indeed, we go along chains. To require *finite* abstract domains would be unnecessarily restrictive. For the natural numbers, for example, one useful (infinite) abstract domain can be described as follows. Coverage of natural numbers is achieved by k *different* natural numbers. For $k = 2$, for example, we take the flat domain with $\perp < x < \top$ for all $x \in \mathcal{N}_0$.

In the same way as the X_i are associated with abstract domains, the $v \in [\![X_i]\!]$ are associated with abstract values $\overline{v}$. We also need to provide abstract domains $\overline{\mathbf{1}}$, $\overline{C_1}, \ldots, \overline{C_m}$ for $\mathbf{1}, C_1, \ldots, C_m$. Finally, abstract variants $\overline{\times}$ and $\overline{+}$ for the combinators $+$ and $\times$ need to be supplied. $\overline{+}$ and $\overline{\times}$ will usually be used in the definition of the $\overline{X_i}$ according to the structure of the equations in the system S.

The definition below is based on the following intuition. The coverage of a test set is computed by taking the supremum of the corresponding abstract values. The commutativity of $\sqcup$ ensures that we are not dependent on the order of the test set elements. In principle, minimal test sets should be enforced. To reflect the incremental generation of a test set based on a current coverage, sequences of test values rather than test sets might be considered. Each element in the sequence should improve coverage. This does not imply that the sequence forms a minimal test set. On the other hand, a subsequent minimalization is always possible.

Definition 1 *Given a finite (test) set $T \subseteq [\![X_i]\!]$, T fully covers X_i, if $\bigsqcup_{v \in T} \overline{v} = \top$. T is a minimal test set if there is no $T' \subset T$ such that $\bigsqcup_{v \in T} \overline{v} = \bigsqcup_{v \in T'} \overline{v}$. A sequence $\langle v_1, \ldots, v_k \rangle$ strictly increases coverage, if $\bigsqcup_{i=1}^{l} \overline{v_i} < \bigsqcup_{i=1}^{l+1} \overline{v_i}$ for $l = 1, \ldots, k-1$.*

We want to stress a further property of the abtraction scheme. Given a set z (for the X_i, the C_i or $\mathbf{1}$) and the corresponding abstract domain $\overline{z}$, abstract values arising from concrete values in z

should exactly correspond to the smallest non-bottom values in $\overline{z}$. For brevity, this requirement is not formalized.

4.3. The fundamental coverage notion

We will present one particular instance of the abstraction scheme corresponding to the fundamental coverage notion aimed for in the paper. Therefore, we need to define the $\overline{X_i}$, the $\overline{C_i}$, and $\overline{\mathbf{1}}$. In the sequel, we assume that $\overline{\mathbf{1}}$ denotes the special poset $\langle\{\bot,\top\},\bot,\top,\{\bot\leq\top\}\rangle$. For the $\overline{C_i}$, there is no generic way to define them. Their definition is usually specific to the equational system at hand and to the particular test scenario. However, there are some common methods to define the $\overline{C_i}$. Often $\overline{C_i}=\overline{\mathbf{1}}$ is appropriate, e.g., for terminals. For finite sets C_i, $\overline{C_i}=\overline{\mathcal{P}}([\![C_i]\!])$, is practical, e.g., for the Booleans. Here, $\overline{\mathcal{P}}(z)$ denotes the poset for the powerset of the set z, i.e., $\langle\mathcal{P}(z),\emptyset,z,\subseteq\rangle$. Methods to separate several equivalence classes are also sensible in many occassions. For integers, for example, it is sometimes useful to consider negative and positive numbers, and 0.

Let us now consider the $\overline{X_i}$. Before we present the ultimate definition, we want to illustrate two extremes. One extreme is the following:

- $\overline{X_i} =^{\text{def}} \overline{\mathbf{1}}$ for $i=1,\ldots,n$
- $\overline{v} =^{\text{def}} \top$ for $v\in[\![X_i]\!]$, for $i=1,\ldots,n$

This means that coverage can be achieved with just one concrete value. It is interesting to notice that by the natural lifting of the coverage notion to the AG level—as discussed in the subsequent subsection—this extreme turns out to correspond to simple rule coverage. There is another extreme, which is not feasible due to infinity of the $[\![X_i]\!]$:

- $\overline{X_i} =^{\text{def}} \overline{\mathcal{P}}([\![X_i]\!])$ for $i=1,\ldots,n$
- $\overline{v} =^{\text{def}} \{v\}$ for $v\in[\![X_i]\!]$, for $i=1,\ldots,n$

Let us look for a more useful definition. There are several ways to derive the $\overline{X_i}$ by observing somehow the structure of the definition in the system S. Figure 3 presents the choice for this paper. The definition uses a special parameter η to keep track of remaining unfolding steps for the X_i. The initial number of unfolding steps (η_0) is configured by some natural numbers λ_i—one for each X_i. As already pointed out in Section 3, $\lambda_i=2$ is usually sufficient. We might indeed want to use different λ_i for the different X_i. This would be useful to put the focus on certain X_i while relaxing the coverage for other X_j.

$$
\begin{array}{l}
\overline{X_i} =^{\text{def}} \mathcal{AT}^S_{\eta_0}(X_i) \text{ for } i=1,\ldots,n, \text{ where}\\
\quad \eta_0:\{X_1,\ldots,X_n\}\to\mathcal{N}_0 \text{ with } \eta_0(X_j)=\lambda_j\in\mathcal{N}_0 \text{ for } j=1,\ldots,n,\\
\quad \begin{array}{lcl}
\mathcal{AT}^S_\eta(\mathbf{1}) & = & \overline{\mathbf{1}}\\
\mathcal{AT}^S_\eta(C_j) & = & \overline{C_j}\\
\mathcal{AT}^S_\eta(X_j) & = & \overline{\mathbf{1}} \text{ if } \eta(X_j)=0\\
\mathcal{AT}^S_\eta(X_j) & = & \mathcal{AT}^S_{\eta[\eta(X_j)-1/X_j]}(e) \text{ if } \eta(X_j)>0 \text{ and } X_j\equiv e\in S\\
\mathcal{AT}^S_\eta(e_1\times\cdots\times e_k) & = & \mathcal{AT}^S_\eta(e_1)\overline{\times}\cdots\overline{\times}\mathcal{AT}^S_\eta(e_k)\\
\mathcal{AT}^S_\eta(e_1+\cdots+e_k) & = & \mathcal{AT}^S_\eta(e_1)\overline{+}\cdots\overline{+}\mathcal{AT}^S_\eta(e_k)
\end{array}
\end{array}
$$

Figure 3: Abstract domains

There are two ways how the X_j are handled in the definition of $\mathcal{AT}^S_\eta$. Either $\eta(X_j) = 0$, then unfolding is stopped, or $\eta(X_j) > 0$, the definition for X_j in S is observed while updating η to decrement the counter for X_j. Decrementing is encoded as updating the function η at X_j as denoted by $\eta[\eta(X_j) - 1/X_j]$. For $\overline{+}$ resp. $\overline{\times}$ we assume an interpretation as (normal) Cartesian product resp. strict product on posets with $\perp$. Recall that the difference between a non-strict and a strict product on posets with $\perp$ is the following. In the normal Cartesian sense, $\perp$ of the product corresponds to the tuple $\langle\perp, \ldots, \perp\rangle$. In the strict case, all tuples containing at least one $\perp$ are unified. This choice is sensible, because the tuples induced by $\overline{+}$ model coverage for the several alternatives in a sum. Some of the alternatives might be covered, others not. On the other hand, the tuples induced by $\overline{\times}$ model coverage for tuples. Non-strict products for the abstract domains do not make sense here because each concrete tuple will immediately cover all components to a certain extent (more than $\perp$).

Note that the defined abstract domains have the finite chain property, since they are constructed from such domains (the $\overline{C_i}$, $\overline{\mathbf{1}}$) just in terms of product operators which preserve the finite chain property. The recursion involved in the definition of $\mathcal{AT}^S_\eta$ is harmless, since the decrementation of η ensures that the number of unfoldings of equations is limited to a finite value. The abstract domains also satisfy the requirement for the abstract values originating from concrete values to be "small" in the sense as explained above.

The definition of the abstract domains is accompanied with the abstraction function for concrete values in Figure 4. The abstraction for the C_i is regarded as given. If we encounter $\mathcal{AV}^S_\eta(v : X_j)$ with $\eta(X_j) = 0$, coverage is trivially satisfied ($\top$). So every value v is fine. Given a tuple, coverage is a tuple, too. Coverage is computed component-wisely. Given a value $\langle v, j\rangle$ from a sum domain arising from the j-th alternative, coverage for the j-th component for the abstract value is equal to the coverage of v. For all the other alternatives, no coverage is achieved ($\perp$). This treatment of sums is useful for a coverage notion, where all alternatives induced by sums should be explored.

$$
\begin{array}{l}
\overline{v} =^{\mathrm{def}} \mathcal{AV}^S_{\eta_0}(v : X_i), \text{ for } v \in [\![X_i]\!], i = 1, \ldots, n, \text{ where} \\
\quad \begin{array}{lcl}
\mathcal{AV}^S_\eta(v : \mathbf{1}) & = & \top \\
\mathcal{AV}^S_\eta(v : C_j) & = & \overline{v} \\
\mathcal{AV}^S_\eta(v : X_j) & = & \top \text{ if } \eta(X_j) = 0 \\
\mathcal{AV}^S_\eta(v : X_j) & = & \mathcal{AV}^S_{\eta[\eta(X_j)-1/X_j]}(v : e) \text{ if } \eta(X_j) > 0 \text{ and } X_j \equiv e \in S \\
\mathcal{AV}^S_\eta(\langle v_1, \ldots, v_k\rangle : e_1 \times \cdots \times e_k) & = & \langle \mathcal{AV}^S_\eta(v_1 : e_1), \ldots, \mathcal{AV}^S_\eta(v_k : e_k)\rangle \\
\mathcal{AV}^S_\eta(\langle v, j\rangle : e_1 + \cdots + e_k) & = & \langle c_1, \ldots, c_k\rangle \text{ where} \\
 & & \quad j \in \{1, \ldots, k\}, \\
 & & \quad c_j = \mathcal{AV}^S_\eta(v : e_j), \\
 & & \quad c_l = \perp \text{ for } l = 1, \ldots, k, l \neq j
\end{array}
\end{array}
$$

Figure 4: Abstract values

4.4. Attribute grammar coverage

Above, we have seen how to construct abstract domains for nonterminals of context-free grammars and abstract domains for the types of attributes. Now, we will join them to abstract domains for nonterminals of attribute grammars.

Let AG be an attribute grammar with context-free base grammar G and attribute types defined by the domain equation system D. In the sequel, we identify G with the equational system corresponding to its productions. $\overline{n}^G$ is used to describe the syntactical coverage of the nonterminal n of G. Therefore, we call it syntactical abstract domain. Similarly, $\overline{T}^D$ for a type T defined in D is used to describe the semantic coverage of attributes of type T. Thus, we call it semantic abstract domain. Now, we build the combined abstract domain for a nonterminal in three steps:

1. Construct a semantic abstract domain $\overline{p}^D$ for each production p of G.
2. Combine the semantic abstract domains for the productions defining a nonterminal n to a semantic abstract domain $\overline{n}^D$.
3. Combine semantic and syntactical abstract domains for each nonterminal n of G to an abstract domain $\overline{n}^{AG}$.

Let p be a production of G and $r_0.a_0 : \mathcal{T}_1, \ldots, r_k.a_k : \mathcal{T}_k$ the attributes associated with p in the attribute grammar AG together with their types. The semantic abstract domain for p is defined as $\overline{p}^D =^{\text{def}} \overline{\mathcal{T}_1}^D \mathbin{\overline{\times}} \cdots \mathbin{\overline{\times}} \overline{\mathcal{T}_k}^D$. The definition means that each attribute has to be covered individually. Furthermore, it reflects the fact that the application of a production in a derivation covers each attribute to a certain extent. If $p_1, \ldots, p_m$ are the productions of G defining the nonterminal n, then we can define $\overline{n}^D =^{\text{def}} \overline{p_1}^D \mathbin{\overline{+}} \cdots \mathbin{\overline{+}} \overline{p_m}^D$. Finally, we define the combined abstract domain $\overline{n}^{AG} =^{\text{def}} \overline{n}^G \mathbin{\overline{\times}} \overline{n}^D$.

5. Test set generation

In this section, we give an algorithm for the generation of test sets. The algorithm basically searches the set of partial derivation trees for—in some sense—smallest test cases. Thus, the particular coverage criterion is fulfilled rather by a larger set of small test cases than a smaller set of large test cases. Since small test cases tend to test the aspects of the descripted language more separately, we gain more information from such test cases for debugging purposes.

5.1. Syntactical rule coverage

Let AG be an attribute grammar with $G = (N, T, s, P)$ as its context-free base grammar. Starting from elementary derivation trees of the form $t_p = p(\mathit{leaf}_{x_1}, \ldots, \mathit{leaf}_{x_m}) \in T_{x_0}(\Sigma)$ for a production $p = x_0 \to x_1, \cdots, x_m \in P$, we can construct all derivation trees using the extension function $\mathcal{E}$. The derivation tree t corresponding to a derivation $s \overset{p_1}{\Rightarrow}_G w_1 \overset{p_2}{\Rightarrow}_G w_2 \overset{p_3}{\Rightarrow}_G \cdots \overset{p_n}{\Rightarrow}_G w_n$ can be constructed in top-down manner as follows:

$$\mathcal{E}(\mathcal{E}(\cdots \mathcal{E}(\mathcal{E}(t_{p_1}, q_1^\star, t_{p_2}), q_2^\star, t_{p_3}) \cdots), q_{n-1}^\star, t_{p_n})$$

for appropriate $q_1^\star, \cdots, q_{n-1}^\star$. Of course, we can also construct t in bottom-up manner:

$$\mathcal{E}(t_{p_1}, q_1'^\star, \mathcal{E}(t_{p_2}, q_2'^\star, \mathcal{E}(\cdots \mathcal{E}(t_{p_{n-1}}, q_{n-1}'^\star, t_{p_n}) \cdots)))$$

for appropriate $q_1'^\star, \cdots, q_{n-1}'^\star$. Actually, t can be constructed in any order of derivation steps.

Every partial derivation tree $t \in T_n(\Sigma)$ can be completed by

- a tree $t_s \in T_s(\Sigma)$, where the only nonterminal leaf in t_s is of sort n, and
- a $t_{n'} \in T_{n'}(\Sigma)$ without nonterminal leaves for every node $\mathit{leaf}_{n'}$ in t.

For the search algorithm used below, we need a measure for the size of a derivation tree t that is strictly increasing with the length of a derivation yielding t. Let us review possible options:

1. the number of non-leaf-nodes of t (this corresponds directly to the length of the derivation yielding t),

2. the number of terminal leaves of t (for complete derivation trees this corresponds to the length of the derived word),
3. the total number of nodes in t, and
4. the sum of 1 and 2.

Unfortunately, the second and third choices are not strictly increasing because productions do not necessarily introduce terminal leaves into the derivation tree, and productions of the form $n \rightarrow \epsilon$ or $n \rightarrow n'$ do not increase the number of nodes. Thus, in the sequel, the size $|t|$ of a derivation tree t refers to 1 or 4.

Definition 2 (smallest terminal derivation tree) *A derivation tree $t \in T_n(\Sigma)$ without nonterminal leaves is called a smallest terminal derivation tree of sort n, if for all derivation trees $t' \in T_n(\Sigma)$ $|t| \leq |t'|$ holds.*

Smallest terminal derivation trees can be calculated searching $T_n(\Sigma)$ starting from $leaf_n$ with a best first search algorithm (cf. [Win92]) using the extension with elementary derivation trees as successor relation.

Definition 3 (context tree) *A derivation tree $t \in T_s(\Sigma)$ is called a context tree of sort n, if t has at least one leaf of sort n. t is called a terminal context tree, if t has exactly one nonterminal leaf.*

Definition 4 (smallest (terminal) context tree) *A (terminal) context tree t of sort n is called a smallest (terminal) context tree, if for all (terminal) context trees t' of sort n holds that $|t| \leq |t'|$.*

As for smallest terminal derivation trees we can use best first search for the calculation of smallest context trees. Combining smallest context trees and smallest terminal derivation trees in an appropriate manner we can construct smallest terminal context trees.

Definition 5 (smallest completion) *A complete derivation tree t is called a completion of a derivation tree t', if there is a subtree t'' of t which can be obtained from t' by replacing the nonterminal leaves from t' by appropriate derivation trees of the corresponding sort. t is called a smallest completion of t', if for all completions t'' of t' holds that $|t| \leq |t''|$. $|t|_{min}$ is the size of a smallest completion of t.*

This definition also makes clear that smallest completions of a derivation tree t can be constructed from t itself, a smallest terminal context tree, and smallest terminal derivation trees for all the nonterminal leaves. Interestingly, we can achieve rule coverage by taking words represented by smallest completions of the elementary trees as test set. However, we should focus on proper words generated by the AG at hand, and not just by its base grammar. Thus, unfortunately, the notion of a smallest completion is not directly useful as is. In our motivating example, if we start from the production for the goto statement, a smallest completion is a derivation tree encoding the following program:

```
begin goto a end.
```

This is not a valid program, since the label `a` is neither declared nor defined.

Definition 6 (correct complete derivation tree) *A complete derivation tree t is called correct, if the equational system on the attributes of t induced by the computations of t has a solution where the conditions of t hold.*

Thus, we have to search for not necessarily smallest, correct completions in a systematic manner. Obviously, the enumeration of all completions, and testing of correctness until a correct completion

$d_{new} := |t|_{min}$
repeat
  $l := [t]; d := d_{new}; d_{new} := \infty$
  repeat
    $ct := head(l); l := tail(l)$
    if ct is complete then
      if ct is correct then
        return ct
      endif
    else
      select a position p of ct for extension
      build the $|.|_{min}$-sorted list l' of all extensions of ct at p
      $l := append(l'_{\leq d}, l); d_{new} := min(d_{new}, |l'_{>d}|_{min})$
    endif
  until $l = []$
until $d_{new} = \infty$
return *fail*

t	derivation tree to be completed
l	list of candidate trees
d	actual tree size limit
d_{new}	limit for the next iteration (the size of the smallest trees not considered so far)
ct	derivation tree currently selected for extension
$l_{\leq d}$	prefix of l containing the elements t with $\|t\|_{min} \leq d$
$l_{>d}$	postfix of l containing the elements t with $\|t\|_{min} > d$
$\|l\|_{min}$	minimum of the $\|.\|_{min}$ of the elements of the list l. If l is the empty list then $\|l\|_{min} = \infty$

Figure 5: Iterative deepening heuristic search

is found suffices. To conclude the example above, a (in a sense smallest) correct completion of the production for the goto statement is the following:

`label a; begin a : goto a end.`

We are now in the position to discuss details of the generation algorithm. Since we have a lower bound for the size of a completion $|.|_{min}$, we can use a heuristic search algorithm like A* (cf. [Bun97, Win92]). Because of the high space requirements of A* the iterative deepening variant IDA* is more practical. The algorithm is sketched in Figure 5.

For each step of the search algorithm we have two ways to extend the current derivation tree t by an elementary derivation tree:

- if t has a leaf of sort n, t can be extended at this leaf by any elementary derivation tree t' of sort n through $\mathcal{E}(t, q^\star, t')$ for an appropriate $q^\star$;
- if t is of sort n, t can be extended at the root by any elementary derivation tree t' with a leaf of sort n through $\mathcal{E}(t', q^\star, t)$ for an appropriate $q^\star$.

The freedom to choose where to extend the current derivation tree can be used to tune the search algorithm. Since we can evaluate computations and conditions in a stepwise manner, derivation trees can be rejected as soon as it becomes clear that a condition fails (or the result of a computation is undefined in the case of partial functions). A possible heuristic for the selection of the extension position (root or one of the nonterminal leaves) is the most-constraint heuristic, i.e., the position with the greatest number of computations and conditions depending on its attributes is selected for extension. This heuristic tries to build the tree in a direction that as many as possible computations and conditions become evaluable as soon as possible.

5.2. Abstract domain coverage

In this section, we assume some definition of the abstract domains $\overline{n}^{AG}$ for (decorated) derivation trees of sort n according to an AG. Actually, we do not rely on the particular definition of $\overline{n}^{AG}$ from Section 4.

Since we only regard test sets of words derived from the start symbol s, we have to extend the abstraction function for sort n to another sort n'.

Definition 7 *Let t be a derivation tree of sort n' and $\{t_1, \ldots, t_l\}$ the set of subtrees of sort n, then* $\overline{t}^n =^{def} \overline{t_1} \sqcup \cdots \sqcup \overline{t_l}$.

Now, we present a simple procedure to construct test sets achieving full coverage (provided it exists). Suppose we want to cover the nonterminal n according to $\overline{n}^{AG}$, we have to generate a test set $T = \{w_1, \ldots, w_l\}$ where $\{t_1, \ldots, t_l\}$ is the set of the derivation trees of words from T and $\overline{t_1}^n \sqcup \cdots \sqcup \overline{t_l}^n = \top$. Starting from $\perp$ as the current coverage C, the search algorithm is applied to generate a correct completion t for $leaf_n$ (which is of course a derivation tree of sort s), and we require that t has to increase the coverage C, i.e., $\overline{t}^n \not\leq C$. The generation step is iterated with $C := C \sqcup \overline{t}^n$ while $C < \top$. For subsequent nonterminals, we start with the coverage reached by the test set generated so far. The generation forces the current coverage to strictly increase. Recall that this does not imply that the generated test set is minimal.

5.3. Guidance of search and termination problems

Due to the structure of the abstract domains we can use the current coverage to guide the search algorithm. As soon as it becomes clear that no completion of a derivation tree will increase the coverage, we remove it from the candidate list. Actually, this situation arises, if the structure of the tree is already covered by the structural part of the current coverage, and the attributes contributing to the semantic part of the coverage criterion are already fully covered or bound to values not increasing the coverage. This way, the search space is more and more reduced with the coverage increasing.

Additionally, we can select as extension position the one with the greatest number of attributes, which contribute to the coverage criterion, depending on its attributes, or which is not yet fully covered in the structural sense. This way, the search algorithm is guided into a direction increasing the coverage as soon as possible. If the current derivation tree is known to increase the coverage but it is not yet complete, the usual search algorithm can be used to find a correct completion.

Because of the attribute computations and conditions not all complete derivation trees are correct. In many cases, full structural coverage with correct derivation trees is not possible. Dually, through structural dependencies, full attribute coverage may be not possible, i.e., in all derivation trees some attributes always take values of a special form. If full coverage is possible, the search algorithm will terminate, since it essentially enumerates the derivation trees. If full coverage is impossible, the search algorithm may not terminate. A guided search as proposed above may reduce the search space to become finite, and thus make the search algorithm terminate.

Due to the expressiveness of AGs, it is in general not decidable, if the search may lead to correct derivation trees improving coverage. Thus, termination can not be guaranteed. There are various ways to recover feasibility of coverage, and thereby to make the search algorithm terminate. It is often possible to refactor the context-free base grammar to better reflect the permitted structure of the words, or to give more precise types to the attributes. In a sense, full coverage needs to be relaxed by excluding certain subsets of the full coverage set explicitly. Ultimately, we can enforce termination by restricting the search depth. For most sensible attribute grammars it should be possible to limit the search depth, e.g., by restricting the number of recursive unfoldings.

5.4. Negative test cases

We have not yet discussed negative test cases so far. They are quite useful in practice in order to test if proper error messages are produced when invalid sentences are processed by a language implementation. There are two kinds of errors to be considered, that is syntactic errors, and violations of context conditions.

We omit a thorough discussion of negative test cases violating the context-free syntax. One option to obtain a context-free grammar useful for the generation of negative test cases is to use an approach known from mutation testing [OL94, DO91]. Let us rather focus on negative test cases which violate context-conditions. It is actually simple to accomplish the generation of such negative test cases in a systematic way.

Let us assume that the computations in an attribute grammar are total rather than partial functions, although is not difficult to lift this restriction. A negative test case can be conceived as a derivation tree where some conditions are not satisfied. The generation algorithm can directly be used if applied to a modified AG with negated conditions compared to the initial AG. Instead of negating conditions, we might also remove conditions, and check afterwards that a generated derivation tree is *not* correct w.r.t. the original grammar.

To avoid confusion of different violations, for a given test set generator pass, only one condition should be negated. There are different coverage criteria which might be useful in the context of an AG with one negated condition. The minimum is, of course, that the production p with the negated condition is covered at all. The coverage notion from Section 4 is useful to enforce different contexts for the nonterminal on the left-hand side of p. There is one specific requirement, which we did not encounter for positive test cases, but which is particularly useful for negative test cases, that is in a given derivation tree, the affected production p should probably covered exactly once. This requirement induces a unique error location.

5.5. Application to the motivating example

Let us consider test set generation for the example AG from Figure 1 and Figure 2. In the generated test programs below we use two additional constructs, namely `skip` as a trivial form of a statement and `true` as a trivial form of a Boolean expression as needed in the if-statement.

Let us consider a test set for the following criteria. In the structural dimension, $\eta_0(n) = 1$ is assumed for all nonterminals n. In the semantic dimension, $\eta_0(T) = 2$ is assumed for all attribute types T. Thereby, all attributes of type $\mathcal{ID\_LIST}$ are enforced to appear in derivation trees in which they get the empty list, a singleton list, and a list with at least two elements as values, if possible. Our implementation of the generation algorithm returns the following test set of 27 programs achieving the greatest possible coverage:

```
begin skip end.
begin if true then skip end.
begin skip; skip end.
begin begin skip end end.
begin if true then skip; skip end.
label a; begin a : skip end.
begin begin skip end; skip end.
label a; begin a : goto a end.
begin begin skip; skip end end.
label a; begin a : skip; skip end.
label a; begin skip; a : skip end.
label a; begin if true then a : skip end.
label a; begin a : if true then skip end.
label a; begin goto a; a : skip end.
label a; begin a : skip; goto a end.
label a; begin a : begin skip end end.
begin label a; begin a : skip end end.
label b; label a; begin b : a : skip end.
label a; begin a : begin goto a end end.
label b; label a; begin a : b : skip end.
label b; label a; begin a : b : goto a end.
begin label a; begin a : skip end; skip end.
label b; label a; begin b : a : skip; skip end.
label b; label a; begin skip; b : a : skip end.
label b; label a; begin if true then b : a : skip end.
label b; label a; begin a : b : if true then skip end.
label b; label a; begin a : b : begin skip end end.
```

For example, test programs are generated where non-local labels are reachable from a block, i.e., the inherited attribute TL of the nonterminal block has to be nonempty. Not that full coverage is not feasible because there are attributes of type $\mathcal{ID\_LIST}$ which cannot be associated with the empty list, e.g., the attribute TL of the left-hand side of rule [ldef].

6. Concluding remarks

First, the results of the paper are concluded.
Second, related work is discussed.
Finally, subjects for future work are indicated.

6.1. Results

The first contribution of the paper is a general and intuitive notion of coverage for attribute grammars. The coverage notion takes into consideration the context-free part and the attributes in an attribute grammar. A trivial instantiation of the coverage notion corresponds to rule coverage. A non-trivial instantiation takes grammar productions and attribute type definitions themselves into account. A certain complexity of derivation trees and attribute values is enforced relying on an unfolding technique to cope with recursion.

The second contribution of the paper is a powerful generation algorithm for test sets which cover a given attribute grammar according to the coverage notion. Only correct decorated derivation trees are generated. Since the generation algorithm relies on some kind of breadth-first search, the generated test cases are in some sense as small as possible. Therefore, redundance is reduced. Termination can not be ensured in general, but means to recover termination have been indicated. Also, non-termination can be observed to a certain extent, if a certain search depth is considered as harmful.

6.2. Related work

In [Pur72], Purdom gives an algorithm to generate a small set of short words from a context-free grammar where each production of the grammar is used in the derivation of at least one word. More advanced coverage notions for context-free grammars are not considered.

There are some attempts to extend Purdom's approach in different ways in order to take context conditions into account. In [CCD+80], a special grammar formalism with actions working on the internal data structures of a generator is used. If during the generation process a violation of the context conditions appears, there are actions to insert text into or delete text from the generated word at dedicated positions. This way, it is guaranteed that the resulting word respects the context conditions.

A more declarative language description is used in [BS82] using context-free parametric grammars. The parameters associated with the grammar symbols can take values from context-free languages. When a parameter value is needed and not yet defined during the generation process, a value has to be generated or provided by the user. Only words derivable with respect to the generated or user-provided parameter values can be generated. A survey of further approaches used in test set generation for compiler testing is given in [Bur94].

The topic of coverage notions for attribute grammars has not been investigated elsewhere. For context-free grammars, Purdom gives just the simple rule coverage. Some authors take static semantics into consideration for the generation of correct programs. Proper coverage criteria are however not involved. Often randomized test set generation is used (cf. [BM83, MS83, BS96]) hoping that the resulting test sets—if large enough—will include all interesting language aspects. In [CD84a, BS94], program instrumentation is used to gather coverage information from prototypical language implementations. Again, coverage notions for declarative language definitions are not developed.

In his book [Jac96], Jack gives a coverage notion and a test set generation algorithm for logic programs based on anti-unification. The main problem with the given form of anti-unification is that sum domains are not treated in a sufficiently precise manner. Two terms with different functors suffice to get full coverage even if there are further functors in the sum. Also, the test sets are generated on a per predicate basis. Translated to the attribute grammar world this means that each nonterminal would have to be tested separately. For attribute grammar implementations, e.g., compilers and interpreters, usually only words derived from the start symbol can be used.

6.3. Future work

We have reasonable experience in generating test sets for simple rule coverage while aiming at correct derivation trees (cf. [Har97, HLR97]). The technique is feasible for non-trivial language definitions. Using the more general coverage notion from this paper blindly, the generated test sets tend to get too complex to be helpful in actual testing. Thus, a primary subject for future work is a feasibility study to apply the technique to a Pascal-like language, and to work out some pragmatic properties of the technique back-to-back. We mentioned already one simple technique to focus on certain aspects of a language definition, that is by enforcing different numbers of recursive unfolding for different nonterminals or attribute types. The ultimate goal in that respect is a test case generation language.

Another subject for future work concerns the termination problems discussed in the paper. A substantial part of the termination problems can be resolved, when the grammar and / or the attribute types are refined to exclude certain cases explicitly. We believe that most reasonable AGs can be completed in this way to achieve normal termination. However, such refinements put a burden on the programmer. Also, it had to be ensured that the grammar refinements preserve the generated language, and that type refinements are sound. One possible direction for improvements is to derive the refinements (e.g., in the form of annotations) automatically by a kind of type inference.

There is a related problem, that is to say, the complexity of the generation algorithm. Especially, in the semantic dimension too little information is used to guide the search discussed in this paper. The conditions and computations are regarded as black boxes. Also, attribute dependencies are not reflected in the coverage notion. A white-box approach, where conditions and computations are specified, for example, as recursive functions, and attribute dependencies are taken into account, could be used to guide the generation algorithm.

Interestingly, also the type inference proposed above, gets more feasible in the white-box setting. Then, it can be inferred, for example, that a function implementing a computation from an AG is only defined on certain parameter patterns. To infer more information than just structural information, dependent types (cf. [XP99, Aug99]) might be useful.

A particular way to implement the white-box approach and to take attribute dependencies into account is based on constraint-logic programming. The computations and conditions are implemented as constraints. We have done very promising experiments in this direction. The constraint system cuts off many parts of the search space, and cases, where full coverage is not possible, are often identified. This approach will be formalized in a forthcoming publication.

Bibliography

[Alb91] Henk Alblas. Introduction to Attribute Grammars. In Henk Alblas and Bořivoj Melichar, editors, *Proceedings of International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, June 1991.

[Aug99] Lennart Augustsson. Cayenne — A Language with Dependent Types. In S. Doaitse Swierstra, Pedro R. Henriques, and Jose N. Oliveira, editors, *Advanced Functional Programming, Third International School, AFP '98*, volume 1608 of *Lecture Notes in Computer Science*, pages 240–267. Springer-Verlag, September 1999.

[Bei90] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, 2nd edition, 1990.

[BM83] David L. Bird and Carlos Urias Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[BS82] Franco Bazzichi and Ippolito Spadafora. An Automatic Generator for Compiler Testing. *IEEE Transactions on Software Engineering*, 8(4):343–353, July 1982.

[BS94] A. Boujarwah and K. Saleh. Compiler test suite: evaluation and use in an automated test environment. *Information and Software Technology*, 36(10):607–614, 1994.

[BS96] Colin J. Burgess and M. Saidi. The automatic generation of test cases for optimizing Fortran compilers. *Information and Software Technology*, 38(2):111–119, February 1996.

[Bun97] Alan Bundy, editor. *Artificial Intelligence Techniques : A Comprehensive Catalogue.* Springer-Verlag, 4th rev. edition, 1997.

[Bur94] Colin J. Burgess. The Automated Generation of Test Cases for Compilers. *Software Testing, Verification and Reliability*, 4(2):81–99, jun 1994.

[CCD+80] A. Celentano, S. Crespi Reghezzi, P. Della Vigna, C. Ghezzi, G. Granata, and F. Savoretti. Compiler Testing using a Sentence Generator. *Software – Practice and Experience*, 10:897–918, 1980.

[CD84a] Z. J. Ciechanowicz and A. C. De Weever. The 'Completeness' of the Pascal Test Suite. *Software – Practice and Experience*, 14(5):463–471, May 1984.

[CD84b] Bruno Courcelle and Pierre Deransart. Proofs of partial correctness for attribute grammars and recursive procedures. Technical Report RR-0322, INRIA Rocquencourt, July 1984.

[CM92] Marc-Michel Corsini and Kaninda Musumbu. Failure Analysis based on Abstract Interpretation. In John Darlington and Roland Dietrich, editors, *PHOENIX Seminar and workshop on declarative programming*, Workshops in Computing, pages 295–309, London, November 18–22 1992. Springer-Verlag.

[DO91] Richard A. DeMillo and A. Jefferson Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[Fos87] Michael J. Foster. Software validation using abstract interpretation. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 2, pages 32–44. Ellis Horwood, 1987.

[Har97] Jörg Harm. Automatic Test Program Generation from Formal Language Specifications. *Rostocker Informatik-Berichte*, 20:33–56, 1997.

[HLR97] Jörg Harm, Ralf Lämmel, and Günter Riedewald. The Language Development Laboratory ($^{\Lambda}\Delta_{\Lambda}$). In Magne Haveraaen and Olaf Owe, editors, *Selected papers from the 8th Nordic Workshop on Programming Theory, December 4–6, Oslo, Norway, Research Report 248*, pages 77–86. University of Oslo, May 1997.

[Jac96] Oliver Jack. *Software Testing for Conventional and Logic Programming.* Number 10 in Programming Complex Systems. Walter de Gruyter, Berlin, 1996.

[JN95] Neil D. Jones and Flemming Nielson. Abstract Interpretation. In Samson Abramsky, Dov M. Gabbay, and Thomas S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4. Clarendon Press, Oxford, 1995.

[Kas80] Uwe Kastens. Studie zur Erzeugung von Testprogrammen für Übersetzer. Bericht 12/80, Institut für Informatik II, University Karlsruhe, 1980.

[LR94] Ralf Lämmel and Günter Riedewald. Provable Correctness of Prototype Interpreters in LDL. In P. A. Fritzson, editor, *Proceedings of Compiler Construction CC'94, 5th International Conference, CC'94, Edinburgh, U.K.*, volume 786 of *Lecture Notes in Computer Science*, pages 218–232. Springer-Verlag, 1994.

[MS83] V. Murali and R.K. Shyamasundar. A Sentence Generator for a Compiler for PT, a Pascal Subset. *Software – Practice and Experience*, 13(9):857–869, September 1983.

[Mye79] Glenford J. Myers. *The Art of Software Testing.* Wiley - Interscience, New York, 1979.

[OL94] A. Jefferson Offutt and Stephen D. Lee. An Empirical Evaluation of Weak Mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, May 1994.

[Paa95] Jukka Paakki. Attribute Grammar Formalisms — A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[Pur72] Paul Purdom. A sentence generator for testing parsers. *Behaviour and Information Technology*, 12(3):366–375, July 1972.

[Ros92] Mads Rosendahl. Strictness Analysis for Attribute Grammars. In Maurice Bruynooghe and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming (PLILP '92)*, volume 631 of *Lecture Notes in Computer Science*, pages 145–157, Berlin, 1992. Springer-Verlag.

[Sud88] Thomas A. Sudkamp. *Languages and Machines.* Addison-Wesley, Reading, Massachusetts, 1988.

[Weg72] Peter Wegner. The Vienna Definition Language. *ACM Computing Surveys*, 4(1):5–63, March 1972.

[Win92] Patrick Henry Winston. *Artificial Intelligence.* Addison-Wesley, Reading, Massachusetts, 3rd edition, 1992.

[XP99] Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *Conference Record of POPL '99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, January 1999. ACM Press.