Grammar Testing

Ralf Lämmel

CWI, Kruislaan 413, NL-1098 SJ Amsterdam
Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam
Email: ralf@cwi.nl
WWW: http://www.cwi.nl/~ralf/

Abstract. Grammar testing is discussed in the context of grammar engineering (i.e., software engineering for grammars). We propose a generalisation of the known rule coverage for grammars, that is, context-dependent branch coverage. We investigate grammar testing, especially coverage analysis, test set generation, and integration of testing and grammar transformations. Grammar recovery is chosen as a subfield of grammar engineering to illustrate the developed concepts. Grammar recovery is concerned with the derivation of a language's grammar from some available resource such as a semi-formal language reference.

1 Introduction

Grammar engineering. Grammars are omnipresent in software development. They are used for the definition of syntax, exchange formats and others. Several important kinds of software rely on grammars, e.g., compilers, debuggers, profilers, slicing tools, pretty printers, (re-) documentation tools, language reference manuals, browsers and IDEs, software analysis tools, code preprocessing tools, and software modification tools. The common practice of grammar development does not quite reflect this importance of grammars. In practice, the most important property of a grammar is to be implementable with a certain tool (e.g., YACC). This focus leads to what we call "grammar hacking", where issues like testing and disciplined adaptation of grammars play a minor role. Companies, which crucially rely on grammars, suffer from this lack of engineering discipline. Grammar hacking now sees a renaissance in the XML-age. We take a different point of view: Grammars are regarded as proper software engineering artifacts (cf. [15,3,12,5] for a similar attitude). We envision *grammar engineering*, where the development, the maintenance, the recovery, and the implementation of grammars are based on well-founded concepts. The *technique* of generalised LR-parsing [14] (as opposed to LALR(1)), for example, covers the full class of context-free grammars. Thereby, it enables a declarative style of grammar specification, and it removes the gap between specification and (YACC-) implementation. Thus, generalised LR-parsing facilitates the *method* of modular syntax definition because the absence of grammar class restrictions allows us to take unions of grammar modules.

Grammar testing. There are several concepts which contribute to grammar engineering. Not all of them have been sufficiently covered in the literature or

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 201–216, 2001.

adopted as common practice. Fields of grammar engineering which are meanwhile reasonably understood are, for example, grammar implementation and grammar reengineering [15]. Fields which lack fundamental underpinnings are grammar adaptation, grammar maintenance, grammar recovery, and grammar testing. The present paper is concerned with grammar testing. From a software engineering perspective, grammars are specifications (describing languages), or programs (serving as compiler compiler input and others). Testing is one standard way to validate specifications or programs (formal analysis is another). The approach to grammar testing described in the sequel covers various technical and pragmatic aspects such as coverage notions, test set generation, correctness and completeness claims for grammars or parsers, and integration of testing and grammar transformations.

Structure of the paper. In Section 2, the contributions of the paper are motivated by a subfield of grammar engineering, namely grammar recovery. In the two subsequent technical sections, grammar recovery serves for illustrative purposes. In Section 3, the notion of context-dependent branch-coverage is developed. It improves on the previously known notion of rule coverage. In Section 4, derived concepts for grammar testing are discussed, e.g., automated coverage analysis and test set generation. In Section 5, the paper is concluded.

Acknowledgement. This work was supported, in part, by *NWO*, in the project "*Generation of Program Transformation Systems*". I am grateful for the comments by the FASE'01 anonymous referees. I am also grateful to Jeremy Gibbons, Jörg Harm, Jan Heering, Paul Klint, Chris Verhoef and Dave Wile for encouraging discussions, and for helpful suggestions. Some results from the present paper have been presented at the 54th IFIP WG2.1 ("Algorithmic Languages and Calculi") meeting in Blackheath, London, April, 3th–7th, 2000, and at the 2nd Workshop Software-Reengineering, Bad Honnef, May, 11th–12th, 2000.

2 Motivation: Grammar Recovery

The following discussion of grammar recovery motivates grammar testing and other grammar engineering concepts. Grammar recovery comprises the concepts involved in the derivation of a language's grammar from some available resource like a language reference, or compiler source code. In [12], a more general account on grammar recovery is given.

2.1 One Scenario

There are several sensible scenarios for grammar recovery. In the present paper, we restrict ourselves to the scenario characterised as follows.

Lack of an approved grammar for a language L. If we want to renovate code in some language L, e.g., in a dialect of an ancient language like COBOL, a suitable grammar is a pre-condition for certain forms of tool support, e.g., grammar-based software modification tools. The compiler vendor might not provide us with the grammar for the language, or the sources for the compiler. Also, the sources

might not be immediately useful for an extraction of the grammar, e.g., in the case of manually implemented frontends. It is also conceivable that the compiler source code and the vendor are not accessible altogether.

Approximative grammar γ_0 for the intended language L. We assume that there is an approximative grammar γ_0 for L. The grammar could be obtained from a semi-formal language reference, or some grammar-based tool the sources of which are available. By *approximative* we mean that we accept that γ_0 is probably incomplete and incorrect w.r.t. L to a certain extent. This is the challenging aspect of the scenario. Thereby, grammar recovery goes beyond grammar reengineering in the narrow sense, where one is merely concerned with the extraction of the base-line grammar from the sources of a reference implementation, and its refactoring [15]. Let us define the terms incorrectness and incompleteness.

Definition 1. *A grammar G* is incorrect w.r.t. an (intended) language L, *if $L \not\supseteq \mathcal{L}(G)$. Here, $\mathcal{L}(G)$ denotes the language generated by G, i.e., all terminal strings derivable from the start symbol of G. The grammar G* is incomplete w.r.t. L *if $L \not\subseteq \mathcal{L}(G)$.*

The definition is inspired by a point of view as favoured in the diagnosis of logic programs (cf. [6]). The problem with the definition is that the intended language L is not directly accessible. Also, we need to know how to locate incorrect or incomplete phrases in the grammar, and how to correct and complete them. Finally, incorrectness and incompleteness are intertwined. An incorrect phrase often causes some incompleteness. Correcting the phrase also contributes to the completion of the grammar. We should point out that our model of correct and complete grammars is a bit naive (for brevity). In practice, one has usually to consider correctness and completeness of *parsers*. Thereby, further requirements might be implied, e.g., suitable and correct parse trees, grammar class restrictions, disambiguation, and others.

Representative trusted test set C written in L. By *trusted* we mean that the code base C is known to comply with L because it is accepted by a reference implementation I such as a parser or a compiler for L. Note that we do not necessarily need I physically. It might also be sufficient if C was approved by someone else who applied I, or by the corresponding standardisation board, or by the compiler vendor. Besides legal issues, it is not clear if re-compilation is a valid option in order to obtain a useful grammar from a compiler or another language implementation, if it is only available in binary form. By *representative* we mean that C should be large enough to experience all possible constructs of the intended language L in some sense.

Derivation of a relatively complete and correct grammar γ_n for L. The overall idea is to parse more and more programs from C, and to resolve the upcoming incompleteness problems indicated by failure of parsing by specific grammar transformations, e.g., by generalising phrases in the grammar accordingly. Due to the intertwined character of incompleteness and incorrectness, the correctness will also be improved in that process because usually the grammar is modified

and not just extended. This stepwise process which results in intermediate grammars $\gamma_1, \gamma_2, \ldots$ is stopped with γ_n if C has been parsed. In each step, a grammar transformation t_i is used to address a particular error or omission. Since we consider γ_0 as an approximation of L, the structure of γ_0 should be preserved in γ_n as much as possible.

2.2 Case Study

We use VS COBOL II as case study. Its grammar was recovered from IBM's reference [10]. As for the test set involved in the project, 2 millions lines of code from several software projects were used. The raw grammar extracted from IBM's reference was corrected and completed by about 300 small transformation steps. The recovered grammar has been published in December 1999 [11]. It is used by COBOL practioners, tool developers, compiler writers and others since then. The grammar is the first publically available quality COBOL grammar. It was obtained in just a few weeks based on a reproducable process for grammar recovery described in [12]. This figure is in strong contrast to other known figures for (unpublished) quality COBOL grammars. It takes some engineering knowledge or a lot of money to come up with good grammars.

From diagrams to the raw grammar. The IBM reference contains large portions of the VS COBOL II syntax in a graphical notation of so-called syntax diagrams. Some diagrams are not even syntactically correct in the sense of the underlying diagram notation. Other diagrams are semantically incorrect in the sense that they do not generate the intended language. Many diagrams are not correct and complete in the sense that the reference uses informal notes to relax or to restrict the diagrams. Using a dedicated parser for syntax diagrams, the raw VS COBOL II grammar was extracted from the reference. This raw grammar provides the initial and approximative grammar γ_0 according to the scenario sketched above. A fragment of the raw VS COBOL II grammar is shown in Fig. 1 using extended BNF notation.

```
Data-description-entry =
  Level-number (Data-name | "FILLER")? Redefines-clause?
  Blank-when-zero-clause? External-clause? Global-clause?
  Justified-clause? Occurs-clause? Picture-clause? Sign-clause?
  Synchronized-clause? Usage-clause? Value-clause?
Redefines-clause =
  Level-number (Data-name | "FILLER")? "REDEFINES" Data-name
```

Fig. 1. Fragment of raw grammar extracted from [10]

Incorrectness. The fragment in Fig. 1 provides a good example for an incorrectness of the raw extracted grammar. The format of `Redefines-clause` does actually cover more than just the structure of the clause itself. We might assume that the correction is performed by a grammar transformation which simply

deletes the obsolete part of the incorrect definition from the raw grammar. The proper definition `Redefines-clause` is the following:

```
Redefines-clause = "REDEFINES" Data-name
```

Error detection by parsing. In general, an incorrectness might go undetected if only parsing is used for diagnosis because incorrectness means that the language generated by a grammar contains words which are not contained in the intended language. The above incorrectness (and many others) will be realised when approved code is parsed, if the code experiences `Redefines-clause`. This is implied by the intertwined character of incorrectness and incompleteness. The incorrect definition of `Redefines-clause` is *in conflict* with the correct form.

Incompleteness. The most basic form of incompleteness is that some nonterminals are not defined altogether. This form of incompleteness can be recognized statically. Otherwise, incompleteness means that certain phrases in the grammar are too restricted, i.e., they do not cover the corresponding construct in full generality. This form of incompleteness can be uncovered by parsing if the test set experiences the corresponding construct as assumed for a representative test set. The IBM reference, for example, contains an informal note that the order of the clauses in a data description entry is immaterial. In fact, actual COBOL code experiences different orders. The sequence of clauses in Fig. 1 should be turned into a permutation phrase.

2.3 Challenges

There are certain questions the answers to which affect the process of grammar recovery in an essential manner. How do we assess the quality of the code base C? How do we precisely know that C is representative? What does it mean for γ_n to be complete and correct? A completeness relative to C can be claimed if $C \subseteq \mathcal{L}(\gamma_n)$, that is, if C can be parsed. According to Definition 1, correctness in a narrow sense means that $L \supseteq \mathcal{L}(\gamma_n)$. However, L is not directly accessible. So what is a realistic requirement for the relative correctness of γ_n? Besides completeness and correctness of γ_n, what is the relationship between γ_0 and γ_n? The latter question is concerned with the stepwise process of correction and completion. In general, we might ask what kind of properties can be required for the transformations t_i for the steps. What does it mean for t_i to correct or to complete resp. the grammar?

In order to answer these questions thoroughly, concepts for testing grammars are worked out in the following two sections. These concepts are not only useful for grammar recovery but also for other grammar-dependent problems, e.g., parser testing, grammar maintenance, and automated software modification.

3 Context-Dependent Branch Coverage

We discuss the notion of context-dependent branch coverage obtained as an essential generalisation of rule coverage [13]. Firstly, rule coverage will be revisited. Secondly, a context-dependent generalisation is developed. Finally, the use of the coverage notion is illustrated in grammar recovery.

3.1 Rule Coverage

Rule coverage simply means that a test set explores all rules of a grammar. It is clear that for each reduced context-free grammar a finite test set achieving coverage of all rules exists. Note that we assume non-ambiguous grammars in the following definition of rule coverage, and also in most subsequent definitions.

Definition 2. *Let $G = \langle N, T, s, P\rangle$ be a context-free grammar. $w \in \mathcal{L}(G)$ is said to* cover *$p = n \to u \in P$ if there is a derivation $s \Rightarrow_G^* x\ n\ y \overset{p}{\Rightarrow}_G x\ u\ y \Rightarrow_G^* w$. $W \subseteq \mathcal{L}(G)$ is said to* achieve rule coverage (RC) for *G, if for each $p \in P$ there is a $w \in W$ which covers p.*

Application to parser testing. Let us provide a scenario where rule coverage can be used in grammar implementation. Consider a parser P which is assumed to implement a grammar G. Let us assume that the implementation even follows the structure of G, e.g., by using recursive-descent parsing. If G is a non-ambiguous grammar, and W is a test set achieving rule coverage for G, then parsing W with P implies that the parser has to experience all rules of G. Major implementational defects, for example, in the sense of incompleteness of P will be detected in this way. These defects will be reported by the failure of P for certain elements from W. The incorrectness of P could be detected using negative test cases covering mutations of G using ideas from mutation testing [9].

G_1	G_2	G_3	G_4
$[r_1]\ s \to A\ B$	$[r_1]\ s \to A\ B$	$[r_1]\ s \to A\ B$	$[r_1]\ s \to A\ B$
$[r_2]\ A \to C\ a$	$[r_2]\ A \to a$	$[r_2]\ A \to C'\ a$	$[r_2]\ A \to C\ a$
$[r_3]\ B \to b\ C$	$[r_3]\ B \to b\ C$	$[r_3]\ B \to b\ C'$	$[r_3]\ B \to b\ C'$
$[r_4]\ C \to \epsilon$	$[r_4]\ C \to \epsilon$	$[r_4]\ C \to \epsilon$	$[r_4]\ C \to \epsilon$
$[r_5]\ C \to c\ C$	$[r_5]\ C \to c\ C$	$[r_5]\ C \to c\ C$	$[r_5]\ C \to c\ C$
		$[r_6]\ C' \to C$	$[r_6]\ C' \to C$

Fig. 2. Sample grammars ($\mathcal{L}(G_1) = \mathcal{L}(G_3) = \mathcal{L}(G_4) \supseteq \mathcal{L}(G_2)$)

Example 1. The test set $\{a\,b\,c\}$ covers all rules of G_1 from Fig. 2 whereas $\{a\,b\}$ does not because rule $[r_5]$ is not covered.

Limitations. It is clear that for any coverage notion, in principle, one can construct grammars G and G' with $\mathcal{L}(G) \neq \mathcal{L}(G')$ so that the difference is not uncovered solely based on test sets achieving coverage. This is implied by decidability results for context-free languages, and also by the fact that test sets have to be finite. In a pragmatic sense, we are looking for a *powerful* coverage notion which is useful in practice to uncover major differences between grammars. Rule coverage is by far not sufficient for that purpose.

Example 2. G_1 and G_2 from Fig. 2 do not generate the same language because G_2 lacks the occurrence of C in $[r_2]$. Note that $\mathcal{L}(G_1) \supset \mathcal{L}(G_2)$ because ϵ is derivable from C. The set $\{a\,b\,c\}$ covers all rules of G_1 and G_2. Thus, rule coverage does not uncover the incompleteness of G_2 w.r.t. G_1.

Let us rephrase the above example in the context of parser testing. Suppose, G_1 serves as specification, but an actual parser accidentally implements G_2. A test set achieving rule coverage for the specification G_1 does not necessarily uncover the incompleteness of the parser implementing G_2. Any testing method as opposed to formal verification is limited in the sense that errors can only be found to a certain extent. The above example illustrates that rule coverage explores a grammar's structure in a rather weak sense.

3.2 Context-Dependent Rule Coverage

We propose a generalisation of rule coverage, where the context in which a rule is covered is taken into account. This idea is easy to formalise.

Definition 3. *Let $G = \langle N, T, s, P\rangle$ be a context-free grammar. If $m \to u\ n\ v \in P$, where $m, n \in N$, $u, v \in (N \cup T)^\star$, then $m \to u\ \boxed{n}\ v$ is called* direct occurrence of n in G. $\mathsf{Occs}(G, n)$ *denotes* the set of all direct occurrences of n in G.

Example 3. All direct occurrences of G_1 from Fig. 2 are listed:

$$\begin{aligned}
\mathsf{Occs}(G_1, s) &= \emptyset \\
\mathsf{Occs}(G_1, A) &= \{[r_1]\ s \to \boxed{A}\ B\} \\
\mathsf{Occs}(G_1, B) &= \{[r_1]\ s \to A\ \boxed{B}\} \\
\mathsf{Occs}(G_1, C) &= \{[r_2]\ A \to \boxed{C}\ a, [r_3]\ B \to b\ \boxed{C}, [r_5]\ C \to c\ \boxed{C}\}
\end{aligned}$$

Definition 4. *Let $G = \langle N, T, s, P\rangle$ be a context-free grammar. $w \in \mathcal{L}(G)$ is said to* cover the rule $p = n \to z \in P$ for the occurrence $m \to u\ \boxed{n}\ v$ *if there is a derivation $s \Rightarrow_G^* x\ m\ y \overset{q}{\Rightarrow}_G x\ u\ n\ v\ y \overset{p}{\Rightarrow}_G x\ u\ z\ v\ y \Rightarrow_G^* w$ with $q = m \to u\ n\ v \in P$. $W \subseteq \mathcal{L}(G)$ is said to* cover $p = n \to z \in P$ for all occurrences*, if there is a $w \in W$ for all occurrences $o \in \mathsf{Occs}(G, n)$ such that w covers p for o. W is said to* achieve context-dependent rule coverage (CDRC) for G, *if W covers all $p \in P$ for all occurrences.*

Example 4. CDRC separates the grammars G_1 and G_2 from Fig. 2. Every test set W achieving CDRC for G_1, e.g., $W = \{a\,b, c\,c\,a\,b\,c\}$ must explore both rules for C in all occurrences of C. Thus, the incompleteness of G_2 w.r.t. G_1 is uncovered.

Theorem 1. *For all reduced context-free grammars $G = \langle N, T, s, P\rangle$, if W achieves context-dependent rule coverage of G, then W also achieves rule coverage of G provided the following side condition holds: $\mathsf{Occs}(G, s) \neq \emptyset$, or s is defined by a single rule.*

Proof. In a reduced context-free grammar, each rule $n \to z \in P$ with $n \neq s$ has at least one occurrence. Thereby, the rule is covered per pre-condition (for even all occurrences). If s is defined by a single rule, this rule will be covered since derivation starts from s. Otherwise, per side condition we know that there are occurrences of s. Then, the rules defining s are covered like all other rules.

Sensitivity. There is a problem with the context-dependent coverage notion as it stands now. The given definition is very sensitive to chain rules and fold/unfold manipulations. These concepts are useful to improve the structure of a given grammar. By *sensitivity* we mean that the fact if a test set achieves coverage or not is dependent on the existence of chain rules, and it varies for grammars which are equivalent modulo fold/unfold manipulations. First, we identify some relevant terms. Then, we indicate a solution for the sensitivity problem.

Definition 5. *The nonterminal n is said to be* non-branching *in a context-free grammar G if there is exactly one defining rule for n in G. $\mathcal{NB}(G)$ denotes the set of non-branching nonterminals in G. A rule is said to be an* injection*, if it is of the form $n \to n'$, where both n and n' are nonterminals. A rule is said to be a* chain rule*, if it is an injection, and the nonterminal on the left-hand side is non-branching.*

Example 5. Although $\mathcal{L}(G_1) = \mathcal{L}(G_3) = \mathcal{L}(G_4)$, CDRC of G_3 can already be achieved with $W = \{a\,b\,c\,c\}$ whereas W is not sufficient for G_1 and G_4. The three grammars are structurally equivalent under chain rule elimination. G_1 does not contain chain rules. G_3 and G_4 contain the chain rule $[r_6]\ C' \to C$. Note that we can also understand the structural differences between the grammars in terms of fold/unfold manipulations. Coverage for G_3 does not imply that the rules defining C are covered for the *two* occurrences of C'. Therefore, coverage is easier to achieve for G_3. The chain rule in G_4 does not affect coverage because C' has only *one* occurrence, and thus the rules for C will be exhausted via the definition of C'.

The sensitivity of context-dependent rule coverage can be decreased considerably, if non-branching nonterminals are handled in a special way. We can consider indirect occurrences as opposed to direct occurrences in Definition 3 where rules of non-branching nonterminals are involved for intermediate derivation steps. We use sequences of direct occurrences in order to represent indirect occurrences.

Definition 6. *Let G be a context-free grammar. The sequence of direct occurrences $\langle n_0 \to u_1\ \boxed{n_1}\ v_1, \ldots, n_{m-1} \to u_m\ \boxed{n_m}\ v_m, \rangle$ for $m > 1$ is called* indirect occurrence of n_m in G reachable via $M \subseteq N$ *if $n_1, \ldots, n_{m-1} \in M$, and $n_1, \ldots, n_m$ are pairwise distinct.*

We can update the notation $\mathsf{Occs}(G, n)$ to refer to both direct and indirect occurrences. Definition 4 is also easy to generalise. We use CDRC^M to denote the refinement of CDRC to take indirect occurrences reachable via M into account. The described sensitivity problem is solved if $\mathcal{NB}(G)$ is chosen for M.

Example 6. Recall the problem from Example 5. The set $\{a\,b\,c\,c\}$ is not sufficient to achieve CDRC* for G_3, although it was sufficient to achieve CDRC.

Note that the simple CDRC is harder to achieve, if non-branching nonterminals are eliminated before coverage is considered. As for chain rules, this normalisation corresponds to chain rule elimination being one of the steps involved in the folklore algorithm for obtaining Chomsky Normal Form. For more general non-branching nonterminals, a simple unfold step is usually sufficient. The reason why we do not attempt such a normalisation is that we want to consider coverage of the original grammar. If some rule is not experienced in a certain context, for example, then, for traceability, the corresponding analysis should refer to the original grammar rather than to a normalised grammar.

Both rule coverage and the context-dependent generalisation of it were stated for pure context-free grammars, that is, basic BNF notation. Often extended BNF (EBNF) notation is used. One way to look at EBNF is that other constructs for branching than just multiple rules for a nonterminal are provided. In this sense, we are looking for slight generalisations of the defined coverage notions, namely branch coverage and the context-dependent generalisation (CDBC) of it. We do not work out these generalisations in the present paper.

3.3 Application to Grammar Recovery

Recall the scenario for grammar recovery from Section 2.2. It is instructive to notice that an uncovered part of an intermediate grammar γ_i, which accepts some code base C' available at this point in the process, provides an indication of either insufficiency of C' *or* incorrectness of γ_i. If full coverage is achieved, we might claim that both the ultimate C has been accumulated, and a correct and complete γ_n has been found. In practice, it is difficult to accumulate a code base which achieves full coverage, at least for a challenging criterion like CDBC. Thus, in a sense the quality of the code base C' and correctness of γ_i are measured in an intertwined manner.

The value of CDBC. The non-trivial coverage notion CDBC as opposed to simple rule coverage adds precision. Consider, for example, the following three rules $A \to B$, $A \to C$, and $A \to D$ defining A. If in some occurrence, A is used where B, C or D is intended instead, a trusted test set cannot cover the corresponding occurrence. Let us consider an example from the VS COBOL II recovery project. In COBOL, data names can be qualified for nested group fields (records). Moreover, a qualification with a file name can be performed. Thus, we have the following syntax:

```
qualified-data-name =
  data-name (("IN"|"OF") data-name)* (("IN"|"OF") file-name)?
```

IBM's reference for VS COBOL II is imprecise about where non-qualified as opposed to qualified data names are to be used. A decent test set will uncover if `data-name` is too specific for some occurrence. CDBC prevents us from generalising the occurrences of `data-name` more than intended.

4 Grammar Testing

Based on the basic notion of context-dependent coverage introduced in the previous section, further concepts for grammar testing can be supplied. In this section, coverage analysis, test set generation, and integration of testing and transformation are discussed. We want to mention that these concepts for grammar testing are certainly also useful for other formalisms than grammars. We can think of, for example, signatures, algebraic data types, or document type definitions in the XML context.

4.1 Coverage Analysis

Given a test set W and a grammar G, a coverage analyser is supposed to return some representation of the coverage of G by W. As for CDBC, we are interested in the branches which are (not) covered for certain occurrences. We will work out the idea of coverage analysis for basic BNF and direct occurrences. As for the representation of coverage, we assume that the coverage of G by W corresponds to a subset of the full coverage set defined as follows.

Definition 7. *Given a context-free grammar $G = \langle N, T, s, P\rangle$, the* full coverage set $\mathcal{FCS}(G)$ for G *is the following:*

$$\mathcal{FCS}(G) = \left\{ \langle p, o\rangle \;\middle|\; \begin{array}{l} p \in P, \\ o \in \mathsf{Occs}(G, n), \textit{where } p \textit{ is of the form } n \to u \end{array} \right\}$$

Derivation of coverage analysers. We use attribute grammars to formalise coverage analysers. Actually, a scheme to derive an attribute grammar $\mathcal{CA}(G)$ implementing the *c*overage *a*nalyser for G is supplied. Essentially, $\mathcal{CA}(G)$ synthesizes a coverage set from a given test set W. Therefore, W is parsed as a list by $\mathcal{CA}(G)$. The context-free grammar underlying $\mathcal{CA}(G)$ is essentially G but with a new start symbol s' to model lists of words in the sense of G:

$$\langle N \cup \{s'\}, T, s', P \cup \{s' \to \epsilon, s' \to s\ s'\}\rangle$$

We want to synthesize an attribute c for all non-terminals to capture the coverage of G by the parsed test set. All nonterminals n (but s') carry an inherited attribute o of type $\mathsf{Occs}(G, n)$ where we consider a special value $\square$ to encode the top level application of rules for s. The following computations are associated with the rules defining the new start symbol s'.

$$\begin{array}{l} s' \to \epsilon \\ \quad s'.c := \emptyset \end{array} \qquad\qquad \begin{array}{l} s'_0 \to s\ s'_1 \\ \quad s.o\ := \square \\ \quad s'_0.c := s.c \cup s'_1.c \end{array}$$

Their interpretation is straightforward. The coverage of the empty list is the "empty coverage", whereas the coverage of a non-empty list is obtained by taking the union of head's and tail's coverage. The computation for the rules in P follow

a common scheme. Given a rule $p \in P$ with $p = l \to x_1 \cdots x_n$ and $l \in N$, $x_i \in N \cup T$ for $i = 1, \ldots, n$, the following computations are associated with p:

$$\begin{aligned} x_{i_1}.o &:= l \to x_1 \cdots x_{i_1-1} \boxed{x_{i_1}} x_{i_1+1} \cdots x_n \\ &\cdots \\ x_{i_m}.o &:= l \to x_1 \cdots x_{i_m-1} \boxed{x_{i_m}} x_{i_m+1} \cdots x_n \\ l.c &:= x_{i_1}.c \cup \cdots \cup x_{i_m}.c \cup \{\langle p, l.o \rangle\} \end{aligned}$$

The $x_{i_1}, \ldots, x_{i_m}$ correspond to the nonterminals on the right-hand side of p. The intention of the computations is simply to propagate the actual occurrence and to combine coverage from all the nonterminals on the right side of a rule while adding the coverage of the particular rule for the inherited occurrence.

Example 7. $\mathcal{CA}(G_1)$ (refer to Fig. 2 for G_1) synthesizes a coverage for $a\,b$ and $c\,a\,b\,c$, where only the element $\langle C \to c\ C, C \to c\ \boxed{C} \rangle$ is missing from the full coverage set $\mathcal{FCS}(G_1)$. Thus, the rule $C \to c\ C$ has not been used for the recursive occurrence of C in $C \to c\ C$.

There are two useful refinements of the above scheme. Firstly, one can count actual applications of rules for the various occurrences. For that purpose, it is sufficient to allow the attributes c to carry multi-sets. Secondly, CDRC^M can be accomplished by propagating indirect occurrences with the attributes o.

4.2 Test Set Generation

A test set generator is a program which computes a test set achieving the desired coverage criterion. Test set generation is useful, for example, in language design, and parser testing. Returning to the application scenario of parser testing in Section 3.1, test set generation automates testing the parser P w.r.t. a reference grammar G.

The generative application of context-free grammars is reasonably understood. Some fundamental algorithms are developed in [13], e.g., shortest derivations to reach a certain nonterminal, and the derivation of tests sets achieving rule coverage. One approach to test set generation is the following. Given a context-free rule, a shortest completion is computed. Thereby, it is possible to compute a small test set of words with short derivations achieving rule coverage. When grammars are applied for the syntax definition of languages, the choice of shortest completions is beneficial for debugging purposes. One can also favour an even smaller test set of words with longer derivations to cover as many rules in one derivation as possible. The context-dependent generalisation of rule coverage does not introduce any complication: Rules occurring in certain contexts are completed instead of just rules.

4.3 Integration with Transformation

Grammars need to be adapted during recovery, maintenance, and elsewhere. Grammar transformations are useful for an operational and formal model of corresponding adaptations. We separate grammar refactorings which do not change

the generated language, and transformations for construction and destruction which go beyond simple semantics-preserving transformations (in terms of the generated language). An application scenario for grammar refactoring is DeY-ACCification and modularisation which are useful in grammar reengineering in order to go from YACC-like pure BNF notation to a richer notation (cf. [15]) including constructs for extended BNF and modules as, for example, in SDF [8]. The steps to correct or to complete a grammar in grammar recovery can be modelled by constructing or destructing transformations. We want to study the relation between grammar transformations and grammar testing.

An operator suite. Let $\mathcal{G}$ denote the set of all context-free grammars. Then, a grammar transformation is a partial function on $\mathcal{G}$. Partial functions have to be considered because of applicability conditions. The operators are explained by describing their effect on an input grammar $G = \langle N, T, s, P\rangle$. Some transformations are applied in a focus $F \subseteq N$, that is, the transformations are supposed to affect only the definitions of nonterminals F. There are the following operators. introduce n as u adds the rule $n \to u$ to G. The operator is applicable if $n \notin N$. fold u to n replaces the phrase $u \in (N \cup T)^\star$ in the rules defining F by n, e.g., $n' \to x\,u\,y$ is turned into $n' \to x\,n\,y$. The operator is applicable if n is a non-branching nonterminal defined by the rule $n \to u$. unfold n replaces the right-hand side occurrences of the nonterminal n in the rules defining F by the definition of n, e.g., $n' \to x\,n\,y$ is turned into $n' \to x\,u\,y$ if G contains the rule $n \to u$. The operator is applicable if n is non-branching in P. eliminate n removes all rules defining the nonterminal n from G. The operator is applicable if n is not reachable from s. The operators for introduction, folding, unfolding, and elimination facilitate grammar refactoring without changing the generated language. The remaining two operators are constructive or destructive resp. in the sense that they increase and decrease of the generated language. include u for n adds the rule $n \to u$ to G. The operator is applicable if n is defined in G, that is, there is at least one rule with n on the left-hand side in G. exclude u from n removes the rule $n \to u$ from G. The operator is applicable if there are two or more rules defining n, one of the form $n \to u$.

Coverage-related relations on grammars. The impact of grammar transformations regarding coverage can be conceived in terms of some relations on grammars, e.g., the property if two grammars are covered by the same test sets.

Definition 8. *We use $\mathcal{R}(G)$ to denote the reduced sub-grammar of G which is obtained by removing rules which are not reachable, or which are not terminated. Given two context-free grammars G and G', and a coverage criterion α, we say* α for G implies α for G' *if for $W \subseteq (\mathcal{L}(\mathcal{R}(G)) \cap \mathcal{L}(\mathcal{R}(G')))$ holds that W achieves α-coverage for $\mathcal{R}(G)$ implies W achieves α-coverage for $\mathcal{R}(G')$. G and G' are called* α-equivalent *if α for G implies α for G' and vice versa.*

Properties of transformations. Based on the above grammar relations, transformations can be characterised w.r.t. coverage as follows.

Definition 9. *A partial function $f : \mathcal{G} \nrightarrow \mathcal{G}$ is said to* improve *(v.s.* hamper*) the coverage criterion α if α for G implies α for $f(G)$ (or vice versa for hampering) for all $G \in \mathcal{G}$ where $f(G)$ is defined. If G and $f(G)$ are* α-equivalent*, f is said to* preserve *α.*

The following theorem states properties of the above operators w.r.t. CDRC^M. The overall conclusion is that refactoring of grammars does not cause sensitivity problems for coverage. The constructing and destructing operators hamper or improve resp. coverage, since they are essentially concerned with adding or removing branches.

Theorem 2.

1. fold u to n *and* unfold n *preserve CDRC*.*
2. eliminate n *and* introduce n as u *preserve CDRC and CDRC*.*
3. include u for n *hampers CDRC and CDRC$^{\mathcal{NB}(G)\cup\{n\}}$.*
4. exclude u from n *improves CDRC and CDRC$^{\mathcal{NB}(G)\cup\{n\}}$.*

Proof. (Sketch)

1. Direct occurrences become indirect occurrences and vice versa.
2. $\mathcal{R}(\cdot)$ neutralizes the eliminated / introduced rule.
3. The included rule has to be covered for $\mathsf{Occs}(G, n)$.
4. The excluded rule has not to be covered anymore.

As for the fold operator and the unfold operator, the consideration of CDRC*, is essential according to the discussion in Section 3.2 (cf. Example 5 in particular). Note that the property of the include operator and the exclude operator to hamper or to improve coverage does not hold for CDRC*. We indeed have to consider indirect occurrences reachable via $\mathcal{NB}(G) \cup \{n\}$ because by including or excluding a rule the defined nonterminal n might change its status to be a non-branching or a branching nonterminal, respectively.

4.4 Application to Grammar Recovery

Approval of uncovered branches. At any point in the process, coverage analysis can be used to compute the coverage of γ_i w.r.t. the currently available and parsed code base C'. Still there is the problem that uncovered branches (in some context) are either an indication of the insufficiency of C' or the incorrectness of γ_i. Let us assume that a reference implementation I for the intended language L is available. Test set generation can be used to generate a test set W from γ_i. We might prefer to generate only programs which improve on the coverage of C'. W can be checked if it is contained in the intended language, i.e., if it is parsed by I. Besides the availability of I, the only pre-condition is that the error messages produced by I are sufficient to separate parsing errors and violations of static semantics. If W is accepted by I, then the phrases corresponding to all branches in all contexts are feasible in L. Thereby, we can approve uncovered branches without even relying on a code base experiencing them.

Correctness and completeness. The above approach suggests a correctness claim w.r.t. CDBC. If I accepts W, the grammar γ_n can be said to be correct w.r.t. CDBC. In this claim, C is not involved. By contrast, completeness is relative to C, that is, γ_n is said to be complete w.r.t. C. In a sense, the ultimate code base C is more important for completing γ_0, whereas test set generation is more relevant to claim correctness of γ_n. Of course, even if I is available to gain confidence in the correctness of γ_n by parsing generated test cases, full correctness cannot be claimed. The property $\mathcal{L}(\gamma_n) \subseteq L$ cannot be checked by repeatedly applying I which models membership test for L. For similar reasons, the completeness claim is inherently relative.

Preservation of structure. Assuming that the extracted raw grammar γ_0 is a useful approximation of L, we want to preserve its structure as much as possible. Both destruction and construction, that is, removal and addition of branches, should be defensive. That is enforced if the transformation sequence $t_1, \ldots, t_n$ satisfies the following requirements:

- Incorrect phrases are removed by destructing transformations.
- Missing phrases are added by constructing transformations.
- Branches of the grammar which can be covered are not removed.
- Added branches are ultimately covered.

The relative completeness and correctness claims for γ_n are strengthened in this way because γ_n can be regarded as a refactored variant of γ_0 with some removed and added branches. The removal and the addition of these branches was solely triggered by C. Context-dependency of coverage is relevant here if branches are to be added or removed only for certain occurrences of a nonterminal.

5 Concluding Remarks

Contribution and applications. Context-dependent branch coverage and derived concepts for coverage analysis, test set generation, and the integration of testing and transformation were developed. These testing concepts are meant as a contribution to grammar engineering. The paper illustrated that more involved coverage criteria than the previously known rule coverage are needed. Our examples were concerned with parser testing and grammar recovery. The paper suggested original relative correctness and completenss claims for parsers and grammars. Several further applications are conceivable, e.g., language design, and grammar minimalisation for automated software renovation.

Related work. Certain rather pragmatic forms of coverage analysis have been suggested by others, e.g., in [2], it was suggested to count rule applications for a YACC-grammar. Previous approaches to test set generation (for testing language processors) are based on either rule coverage or randomized test sets (cf. [1]). Burgess [4] compiled a survey on compiler testing. A challenging problem in testing compilers or other language processors is the generation of semantically correct programs. Thereby, generation of test sets and feasibility of coverage is

considerably more complicated. One can think of coverage in two dimensions for the language definitions specified, for exampple, with attribute grammars: a syntactical dimension corresponding to the underlying context-free grammar, and a semantical dimension corresponding to the attributes, their types, and the computations associated with the productions. This issue is examined in some depth in [7] based on a related coverage criterion.

Perspective. We are working on adequate tool support for coverage analysis, coverage visualisation, test set generation, and grammar transformation. This project is challenged by the fact that the tools should scale up for complex grammars and huge test sets. We would also like to apply our concepts to document type definitions in the XML context. At the conceptual level, there are the following directions for future work. Our relative notions of grammar correctness and completeness can definitely be further improved or complemented. One would like to consider metrics, for example, to quantify the structure preservation for the raw grammar γ_0. Our relative correctness claim for γ_n was based on the generation of *positive* test cases from the grammar to see if they are *parsed* by a reference implementation. Dually, we could use *negative* test cases, which had to be *refused* by the reference implementation, for a stronger completeness claim. Negative test cases could be obtained via mutation testing [9].

References

1. D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.
2. 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.
3. M. Brand, M. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In S. Tilley and G. Visaggio, editors, *Proceedings of the sixth International Workshop on Program Comprehension*, pages 108–117, 1998.
4. C. J. Burgess. The Automated Generation of Test Cases for Compilers. *Software Testing, Verification and Reliability*, 4(2):81–99, jun 1994.
5. M. de Jonge and J. Visser. Grammars as Contracts. In *Proc. of GCSE 2000*, LNCS, Erfurt, Germany, 2001. Springer-Verlag. to appear.
6. G. Ferrand. The Notions of Symptom and Error in Declarative Diagnosis of Logic Programs. In P. Fritzon, editor, *Automated and Algorithmic Debugging, First International Workshop, AADEBUG'93*, volume 749 of *Lecture Notes in Computer Science*, pages 40–57. Springer-Verlag, 3–5 May 1993.
7. J. Harm and R. Lämmel. Two-dimensional Approximation Coverage. *Informatica*, 24(3), 2000.
8. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
9. W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.
10. IBM Corporation. *VS COBOL II Application Programming Language Reference*, 1993. Release 4, Document number GC26-4047-07.

11. R. Lämmel and C. Verhoef. VS COBOL II Grammar Version 1.0.3. `http://www.cwi.nl/~ralf/grammars`, 1999–2001.
12. R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. Submitted, available at `http://www.cwi.nl/~ralf/`, July 2000.
13. P. Purdom. A sentence generator for testing parsers. *BIT*, 12(3):366–375, 1972.
14. J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.
15. M. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions. In J. Ebert and C. Verhoef, editors, *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 151–160. IEEE Computer Society, March 2000.