EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Black-box random testing of the mCRL2 toolset using attribute grammars

Geelen, M.P.H.

Award date:
2014

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Black-box Random Testing of the mCRL2 Toolset Using Attribute Grammars

Mark Geelen, BSc

January 7, 2014

Master's Thesis
Computer Science and Engineering

Supervisors:
J.W. Wesselink, PhD
T.A.C. Willemse, PhD

Eindhoven University of Technology
Department of Mathematics and Computer Science

Abstract

This research presents a grammar-based approach to random testing of the mCRL2 toolset in order to find previously undiscovered errors. The testing method is black-box and we design test cases where other tools from the toolset are used for verification. This approach avoids the re-implementation of the specification of the tools to serve as a testing oracle. The input and output of the tools adhere to context-free grammars. We enhance these grammars with attributes in order to ensure the generation of strings that are correct with respect to various static semantics. We investigate how we can model semantic constraints using these attribute grammars. To this end, we take a random string generation algorithm and modify it to incorporate these attributes. The presented method is applied to find errors in the mCRL2 tools. The method is successful in finding known errors in older versions. The method has not yet been successful in discovering new errors.

Contents

1 Introduction **4**

2 Related work **6**

3 Background **7**
3.1 Context-free Grammars . . . 7
3.1.1 Ambiguity . . . 8
3.1.2 Tokens . . . 9
3.1.3 Extended BNF . . . 9
3.2 Boolean Equation Systems . . . 9

4 String Generation **12**
4.1 Length-based Algorithm . . . 13
4.1.1 Limitations . . . 14
4.1.2 Weighting . . . 15

5 Context **17**
5.1 Meaningful Terms . . . 17
5.2 Attribute Grammars . . . 17
5.3 Length . . . 22
5.3.1 Termination . . . 23
5.4 Traversal Strategies . . . 25

6 Constraints **26**
6.1 Variable Binding . . . 26
6.2 Monotonicity . . . 27
6.3 Operator Precedence . . . 29
6.4 Typing . . . 29
6.5 Guarded Processes . . . 31

7 Directed Testing **34**

8 Test Case Design **35**
8.1 Partial Testing . . . 35
8.2 Execution . . . 37

9 Implementation **39**

10 Experimental Setup **41**
10.1 Validation . . . 41

11 Results **45**
11.1 Validation . . . 45
11.2 New Errors . . . 47
11.3 Directed Testing . . . 50

12 Conclusion **51**

13 Future Work **52**
13.1 Method . . . 52
13.2 Experimentation . . . 52

Bibliography **53**

A Context-Free Grammars **54**
A.1 Boolean Equation System 54
A.2 Data Expressions 54
A.3 Parameterized Boolean Equation System 54
A.4 mCRL2 Process Specification 54

1 Introduction

The purpose and importance of software testing is widely known. There are several flavors of software testing like the basic unit testing and regression testing, where previously occurred errors serve as test cases for future versions of the software. For large software systems where correctness is critical, like the mCRL2 toolset, testing is an important aspect of the development and maintenance of the system. Ideally, the system is tested for every possible input, by which the system can be made error-free. However, the input domain can be too large to make this feasible. This is where random testing poses a solution: if the domain is large and we have an efficient way to generate (a lot of) random input, random testing is an effective testing approach ([4]).

The mCRL2 toolset[1] consists of algorithms (tools) that work with the mCRL2 formal specification and modeling language ([3]). The language consists of a process algebra with which behaviour of systems and protocols can be modelled. The toolset allows for formal modeling, analysis and verification of systems and protocols. Furthermore, it can be used to simulate concurrent processes and visualize the state space. Specifications can be extended with data, i.e. abstract data types and higher order functions.

The mCRL2 toolset employs six formalisms: Boolean equation systems (BESs), parameterized Boolean equation systems (PBESs), mCRL2 process specifications, linear process specifications (LPSs), labelled transition systems (LTSs) and mu-calculus formulas. The tools in the toolset operate on various types of input, which is a term from the language of these formalisms. There is for example a tool that can transform an mCRL2 specification into a linear process specification. Likewise, there are tools that operate on Boolean equation systems, labelled transition systems, etc. These tools consist of implementations of various algorithms and most have undergone years of development. However, partly due to the size of the system and the amount of code, these tools are not error-free. Errors are encountered every now and then through usage of the toolset, when a user receives unexpected output for their input.

The main goal of this research is to find errors in the tools of the mCRL2 toolset. We can identify three subgoals:

1. provide a method to design test cases for the black-box testing of tools from the mCRL2 toolset
2. provide a method to generate valid input for the tools
3. apply the methods to the mCRL2 toolset to find errors

The language of the formalisms are defined through context-free grammars and because of this a grammar-based approach is justified. This grammar-based approach provides an efficient way to generate a large amount of random terms, which motivates the choice for random testing. Moreover, we are interested in black box testing: we either have a specification or an invariant of the functionality of each tool, but assume no knowledge on the implementation. So we will randomly generate input and verify the output of a tool using its specification or an invariant.

It is infeasible and sometimes impossible to re-implement the specifications of the tools to serve as test oracles. To overcome this, static, stand-alone test cases are devised in which other tools from the mCRL2 tool set are used to verify the results of a test. As a general example, assume tool A that doubles a natural number and B that decides whether a natural number is even. If we run B on the output of A, then B should always say yes. So a test case would be to randomly generate input for A and use B on the output. If B ever says no, there is an error in either of the tools and we have the inputs to serve as a counter-example.

[1] http://mcrl2.org

To generate input for test cases, we choose an algorithm that operates on context-free grammars to produce random terms from these grammars. However, a lot of input that we generate directly from a context-free grammar is not useful. Semantic requirements like well-typedness will render most generated terms useless. There are several types of static semantics that an input term is required to adhere to and a randomly generated syntactically correct string only has a small chance to be semantically correct. To force the generation of meaningful strings we impose constraints on the generation algorithm by enhancing the context-free grammar with attributes. Attribute grammars are commonly used in compiler technology and we will use it in a similar manner, although the generation of semantically correct strings differs from recognizing them, which is the common application of attribute grammars in compilers.

To determine the effectiveness of the presented method, we apply it to previous versions of the toolset of which several errors have been identified by previous random testing. The random testing method should be able to find these errors.

This thesis will start off by discussing literature that relates to this research. Some background information on context-free grammars will be presented, leading to an algorithm that produces a term from this grammar. The subsequent section will describe and define attribute grammars, after which they are used to design constraints. Constraints are presented for several semantic requirements. Directed testing will be described and the thesis continues with a description of how test cases are designed and executed. The implementation is discussed after which the setup for the experiments is described. The results of the experiments are presented and followed with the conclusions and future work.

2 Related work

The research described in this thesis relates to concepts involving the automatic generation of test data. In [5], several approaches to the automatic generation of test data are described and compared. Random testing is discussed along with grammar based testing. These approaches are considered useful in cases where there is no knowledge of the programs under test, except for their input language.

Ralph Lämmel ([8]) presents an algorithm that features full combinatorial coverage. This means that from a context-free grammar, all strings in the grammar's language are generated. To prevent the generation of the entire (possibly infinite) language, control mechanisms can be used to limit the set of strings generated. However the main purpose is still the generation of a set of terms and this differs from our goal in employing an algorithm that in fact generates a single string from the language at random. Furthermore, the method does not provide means to add constraints to fulfill semantic requirements.

We aim to generate strings that are valid with respect to various static semantics. For this reason, we consider related work involving string generation using context-sensitive grammars. Several domain specific languages (DSL) are designed that use attribute grammars for string generation. One of the most powerful DSLs is the Data Generation Language (DGL) from Peter Maurer ([9]). Grammar rules are written directly in this language and variables can be used to store context in order for generated strings to adhere to certain semantic requirements. The DGL also facilitates weighting of production rules. Unfortunately, the production rules and constraints have to be combined, decreasing the usability.

Duncan and Hutchinson present a DSL in [2] for the generation of strings using attribute grammars. One of the advantages to this DSL is the support for EBNF notation in the grammars. The authors combine the notions of inherited and synthesized attributes into *hybrid* attributes. This is a useful simplification when both types are used. However, in the case where e.g. only inherited attributes are needed, this combination is redundant and arguably more complex. Furthermore, it forces a certain traversal strategy which means certain semantic requirements cannot be fulfilled. The authors present the concept of *guards*, similar to conditions in compiler technology, which guide the selection of production rules. We will use the same concept of guards in the definitions and algorithms pertaining to attribute grammars.

Lava ([16]) is another DSL that facilitates string generation with attributes. One downside to Lava is that it features no generic approach to add constraints to a grammar under test. However, the modularity and separation between grammar and constraints are very useful from a tester's perspective. We will use a similar approach when implementing the presented method, labelling production rules in the grammar and letting constraints refer to these productions using the labels.

Norman Paterson's work ([11]) is an attribute grammar-based approach to genetic programming. It contains a useful comparison between two common context-sensitive grammars, namely two-level grammars and attribute grammars, for use in string generation. Furthermore, it does not force a certain traversal strategy but does not provide a method to design constraints.

In [1] presents an approach similar to this research. This research presents a method for the random generation of Erlang programs for use in random testing. The approach is attribute grammar-based and is implemented in a DSL style format. One downside is the mixing of constraints and context-free grammars in the same notation. When a user wants to reuse a constraint for different grammar rules, this constraint has to be added to both sets of rules separately.

3 Background

3.1 Context-free Grammars

The input and output of tools from the mCRL2 toolset are strings from languages defined by context-free grammars. A grammar is a set of rules that specify how sentences or phrases can be formed by applying these rules to the symbols in the grammar. In context-free grammars there is no context, which means the rules can be applied to the symbols regardless of the context. From the book of Michael Sipser [15] we can get a formal definition for a context-free grammar, which will be used to explain the concepts of a context-free grammar as they are used in the remainder of this document.

Definition 3.1.1 (Context-free grammar). A context-free grammar is a four-tuple $G = (V, \Sigma, R, S)$ where

- V is the set of non-terminals
- Σ is the set of terminals
- R is the set of production rules of the form $V \to (\Sigma \cup V)^*$
- $S \in V$ is the start symbol

The * symbol is the Kleene star, so an element out of the set $(\Sigma \cup V)^*$ is any sequence of elements from Σ and V. For example, if $a, b \in \Sigma$ and $N \in V$, then $abNba \in (\Sigma \cup V)^*$. If there are multiple production rules with the same left-hand side non-terminal, we can use a shorthand notation using | to separate the alternatives, i.e.

$$\begin{matrix} S \to a \\ S \to b \end{matrix} \quad \text{and} \quad S \to a \mid b \tag{3.1}$$

represent the same set of production rules.

Given a context-free grammar $G = (V, \Sigma, R, S)$, we want to say that a term $v \in (\Sigma \cup V)^*$ can be derived from another term $u \in (\Sigma \cup V)^*$ if there is a production rule $\alpha \to \beta \in R$, that if applied to u, yields v. The application of a production rule $\alpha \to \beta$ to u involves replacing an occurrence of α in u by β. Formally, we can define the binary relation $\Rightarrow_G$ on elements of $(\Sigma \cup V)^*$ as follows

$$(u \Rightarrow_G v) \Leftrightarrow (\exists \alpha \to \beta \in R : (\exists u_1, u_2 \in (\Sigma \cup V)^* : u = u_1 \alpha u_2 \wedge v = u_1 \beta u_2)) \tag{3.2}$$

If a term $u \in (\Sigma \cup V)^*$ can be derived in one or more steps from the start symbol, then it is a so called *sentential form.* A *string* is a sequence of terminal symbols, so an element $\sigma \in \Sigma^*$. The language of a context-free grammar is all sentential forms that are strings. So the language is a set of strings that can be derived from the start symbol, formally defined as follows.

Definition 3.1.2 (Language). The language of grammar G, denoted as $L(G)$ is the set

$$\{w \in \Sigma^* \mid S \Rightarrow_G^* w\} \tag{3.3}$$

where $\Rightarrow_G^*$ is the reflexive transitive closure of $\Rightarrow_G$.

For the remainder of this document the subscript $_G$ is omitted when there is no confusion about the grammar used. Example 3.1.1 shows how a context-free grammar defines a language.

Example 3.1.1. *The context-free language $\{a^n b^n \mid n \geq 1\}$ can be made through the following two grammar rules.*

$$S \to aSb \mid ab \tag{3.4}$$

With start symbol S, we can apply the first production rule $n-1$ times for a given n, after which we can apply the second production rule.

$$\begin{gathered} S \\ \Rightarrow \\ aSb \\ \Rightarrow \\ aaSbb \\ \Rightarrow \\ aaabbb \end{gathered} \tag{3.5}$$

If we take $n = 3$, the above derivation results in the string $aaabbb$ or a^3b^3.

A *leftmost derivation* is a derivation in which each production rule is applied to the left-most non-terminal. A parse tree or *derivation tree* ([13]) is a tree-like representation of a derivation of a string. In a derivation tree, the root is the start symbol, the leaves are terminal symbols and each node represents the application of a production rule. Figure 3.1 shows the derivation tree for example 3.1.1.

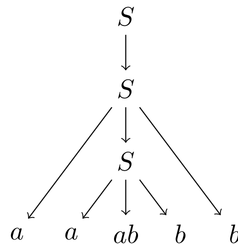


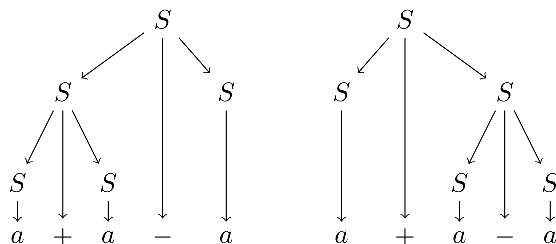
Figure 3.1: Derivation tree for example 3.1.1

The leaves of a derivation tree represent the terminal string if the symbols from the leaves are put in sequence, in the order in which they are encountered in a left-to-right traversal of the tree.

3.1.1 Ambiguity

A different (order of) application of production rules can produce the same terminal string. A string is called *ambiguous* when there are two different leftmost derivations. This means there exists two different sequences of production rules that, when continuously applied to the left-most non-terminal, produce the same terminal string.

Example 3.1.2. *Consider a context-free grammar with production rules $S \to S + S \mid S - S \mid a$. The string $a + a - a$ is ambiguous in this grammar as it has two different derivation trees.*



Ambiguity can pose a problem when considering the semantics of a string from a language as an ambiguous string can have different interpretations. When grammars containing ambiguities are used in practice, they are usually extended with a notion of rule precedence. With rule precedence, priorities are assigned to production rules in order to resolve ambiguities. When two production rules can be applied, the highest priority gets precedence, so there is no ambiguity.

The context-free grammars in mCRL2 contain these precedence rules. For generating a string from a context-free grammar ambiguities can be ignored. For the semantic requirements that are defined using the grammars, we assume the existence of these precedence rules in the definitions, such that there are no ambiguities. In 6.3 we will see a constraint that disambiguates a context-free grammar using rule precedence.

3.1.2 Tokens

Context-free grammars often contain a notion of variable names or other alphanumeric constants. For example, a natural number can be represented in a context-free grammar with a recursive production rule that produces one or more digits. However, we would like to abstract from these constants, which we designate as tokens. Tokens are non-terminals for which we do not include any production rules. Instead, we opt to define the set of allowed string values for each token separate from the grammar.

3.1.3 Extended BNF

Definition 3.1.1 is in Backus-Naur form. There is an extension to BNF (Extended BNF or EBNF [20]), that allows for explicit descriptions of repetitions and choices of production rules, as well as nesting right-hand side alternatives. The relation between EBNF and BNF is described in [20] and there is a straightforward transformation from EBNF to BNF which involves replacing repetitions by recursions and choices by two alternative production rules where one of them is the empty string.

The context-free grammars in mCRL2 are in EBNF. We require a preprocessing step transforming the grammars into BNF, before applying the string generation algorithm to these grammars. The following section describes the approach for generating a string from a context-free grammar.

The following section illustrates the concepts of the context-free grammar as used in this research in a formalism used in mCRL2, namely *Boolean equation systems*. An example use of a token is also described.

3.2 Boolean Equation Systems

One of the formalisms defined by a context-free grammar used in the mCRL2 set is the formalism of Boolean equation systems. A model checking problem can be translated to a Boolean equation system (BES) [18] which is a series of fix-point equations ranging over the Boolean lattice. Example 3.2.1 shows a BES consisting of two fix-point equations.

Example 3.2.1. *The following BES contains two propositional variables and two fix-point equations with fix-point operators μ and ν, the least- and greatest fix-point respectively.*

$$\begin{array}{c}(\mu X_1 = X_1 \wedge X_2)\\(\nu X_2 = true)\end{array} \tag{3.6}$$

Boolean equation systems will serve as a primary example in this document when explaining the concepts and methods used. The BES language can be defined through the following context-free grammar.

Definition 3.2.1 (**CFG Boolean equation systems**).

$$
\begin{array}{rcl}
BES & \rightarrow & \mathcal{E} \\
\mathcal{E} & \rightarrow & (\sigma X = \phi) \mid (\sigma X = \phi)\mathcal{E} \\
\phi & \rightarrow & \phi \Rightarrow \phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid true \mid false \mid X \\
\sigma & \rightarrow & \mu \mid \nu
\end{array}
\tag{3.7}
$$

In this grammar, X is a token for propositional variables. We consider the set $\{X_i \mid i \geq 1\}$ as possible token values for X. The following example illustrates how a terminal string from the language defined through the context-free grammar can be built using production rules. Every application of a production rule on a sentential form produces a new sentential form and eventually a terminal string.

Example 3.2.2. *Starting with the start symbol BES, we can apply production rules from the BES grammar to produce the BES from example 3.2.1 as follows:*

$$\begin{gathered} BES \\ \Rightarrow \\ \mathcal{E} \\ \Rightarrow \\ (\sigma X_1 = \phi)\mathcal{E} \\ \Rightarrow \\ (\mu X_1 = \phi)\mathcal{E} \\ \Rightarrow \\ (\mu X_1 = \phi \wedge \phi)\mathcal{E} \\ \Rightarrow \\ (\mu X_1 = X_1 \wedge \phi)\mathcal{E} \\ \Rightarrow \\ (\mu X_1 = X_1 \wedge X_2)\mathcal{E} \\ \Rightarrow \\ (\mu X_1 = X_1 \wedge X_2)(\sigma X_2 = \phi) \\ \Rightarrow \\ (\mu X_1 = X_1 \wedge X_2)(\nu X_2 = \phi) \\ \Rightarrow \\ (\mu X_1 = X_1 \wedge X_2)(\nu X_2 = true) \end{gathered} \tag{3.8}$$

Here, production rules are applied continuously to the left-most non-terminal.

Several tools from the mCRL2 tool set under test operate on Boolean equation systems. These tools contain implementations of various algorithm for solving or manipulating a BES. To test these implementations we will generate random BESs as input for these tools.

4 String Generation

The tools in the mCRL2 tool set that we aim to test all require a term from one of the mCRL2 formalisms as input. Since these formalisms are defined through context-free grammars, we can use a generic approach to generating a string from a context-free grammar and apply this to the context-free grammars in mCRL2.

Given a context-free grammar G, we seek an algorithm that produces a string $\sigma \in L(G)$ at random. A naive top-down approach to generate a string from $G = (V, \Sigma, R, S)$ would be the following.

Algorithm NaiveTopDown(V, Σ, R, S)

1 $u \leftarrow S$
2 **while** there are non-terminals in u
3 **do** take the leftmost non-terminal N
4 choose production rule with N as its left-hand side
5 apply this production rule to u
6 **return** u

So starting with the start symbol, the naive top-down generation algorithm will choose random production rules to apply until it ends up with a terminal string. However, there is an issue with this naive approach: the algorithm has the potential to go on indefinitely. If the continuous application of production rules does not eventually reduce the number of non-terminals to zero, the algorithm will not terminate. So the choice of production rules has to be guided such that we eventually end up with a terminal string.

Example 4.0.3. *Consider a context-free grammar with start symbol S and the following production rules:*

$$S \rightarrow bSS \mid a \tag{4.1}$$

Using the naive algorithm with this grammar and assuming the algorithm chooses uniformly at random among the alternatives, the probability that u is a terminal string after applying one of the production rules is $1/2$, in which case the result is a. The probability that u is a terminal string after any number of applications is $1/2 + 1/8 + 1/16 + ...$ which converges to approximately 0.874. This shows that termination is not guaranteed for this naive algorithm.

There is need for a control mechanism: a certain metric of strings that can be set to a limit and an algorithm that chooses production rules such that this limit is not exceeded. Several control mechanisms are identified and defined in [21]. The most commonly used mechanism is *depth control*, which is a limit to the height of the resulting derivation tree. Depth is a simple and intuitive metric.

Another simple metric is the *length* of a string. The length of a sentential form u is the amount of symbols in u: non-terminals, tokens and terminal symbols have a length of one, so the length of a terminal string is the amount of terminal symbols in that string. The algorithm from Bruce McKenzie [10] can generate a string of length n from a context-free grammar uniformly at random, meaning every string of length n in the language is equally likely to be generated. Having an algorithm for which the coverage is known is a motivation for length based control.

The algorithm with length control will make sure the length of the generated string is equal to a certain length limit. For a given length n, the algorithm will only apply production rules that can produce an n-length string. Given a length n and a sentential form u, we choose lengths for every non-terminal in u, dividing n among the non-terminals. This guarantees termination. In the following sections a more detailed description of the algorithm and its limitations will be given.

4.1 Length-based Algorithm

This section describes the algorithm from Bruce McKenzie ([10]), which can be used to generate a string from a context-free grammar while imposing length control. We use index notation used in [10] for the elements of a context-free grammar. So we denote the set of non-terminals as $V = \{N_1, N_2, ..., N_r\}$ where r is the number of non-terminals. The set of production rules R is a mapping from non-terminals to terms from $(\Sigma \cup V)^*$. The set R can be written as follows.

$$R = \{\pi_{ij} : N_i \to x_{ij1}x_{ij2}...x_{ijt_{ij}} \mid 1 \leq i \leq r \wedge 1 \leq j \leq s_i\} \tag{4.2}$$

Here, s_i is the amount of production rules with N_i as the left-hand side and t_{ij} is the amount of symbols in the right-hand side of production rule π_{ij}. We use π_{ij} as a shorthand notation or 'name' for the production rule $N_i \to x_{ij1}x_{ij2}...x_{ijt_{ij}}$. The right-hand side $x_{ij1}x_{ij2}...x_{ijt_{ij}}$ is an element from $(\Sigma \cup V)^*$.

The algorithm will always produce a string of length n, by disallowing production rules with which no string of length n can be built. This control mechanism ensures termination. Furthermore, the algorithm makes choices in production rules irrespective of the grammar structure. So for two different grammars that define the same language, the same string has the same probability of being generated. This is true for an unambiguous grammar. This limitation is described in section 4.1.1.

The algorithm uses a few helper functions that return the amount of strings of certain length a production rule can generate. These functions are used to calculate the 'weights' that will be assigned to each production rule. The function $f_i(n)$ is a list of integers, which represents the number of strings of length n that can be built using production rules with N_i as the left hand side non-terminal.

$$f_i(n) = [\ sum(f'_{ij1}(n)) \mid 1 \leq j \leq s_i\ ] \tag{4.3}$$

For every production rule with N_i as the left hand side non-terminal, there is a list entry in $f_i(n)$. The *sum* function is a straightforward summation on the list elements.

$$sum([l_1, l_2, ...l_m]) = \sum_{i=1}^{m} l_i \tag{4.4}$$

The function $f'_{ijk}(n)$ returns the number of strings of length n that can be made out of the symbols $x_{ij1}...x_{ijt_{ij}}$, which make up the right hand side of the production rule $N_i \to x_{ij1}...x_{ijt_{ij}}$. So we have t_{ij} as the amount of symbols in α_{ij}, the right hand side of the production rule π_{ij}. The function can be recursively defined, processing the first symbol continuously. For $k < t_{ij}$, the function is defined as follows.

For $n = 0$:

$$f'_{ijk}(0) = [] \tag{4.5}$$

For $k \neq t_{ij} \wedge n \neq 0$:

$$f'_{ijk}(n) = \begin{cases} [sum(f'_{ij(k+1)}(n-1))] & \text{if } x_{ijk} \in T \\ [sum(f_{x_{ijk}}(l)) \times sum(f'_{ij(k+1)}(n-l)) \mid 1 \leq l \leq n - t_{ij} + k] & \text{otherwise} \end{cases} \tag{4.6}$$

For $k = t_{ij} \wedge n \neq 0$:

$$f'_{ijt_{ij}}(n) = \begin{cases} [sum(f_{x_{ijt_{ij}}}(n))] & \text{if } x_{ijt_{ij}} \notin T \wedge n > 1 \\ [1] & \text{if } x_{ijt_{ij}} \in T \wedge n = 1 \\ [0] & \text{otherwise} \end{cases} \tag{4.7}$$

The complete algorithm is listed below, rewritten slightly from [10] to make it more compact. In the algorithm we abstract from the grammar as a parameter. The algorithm operates on a symbol x_{ijk} from production rule π_{ij} and returns a terminal string. The variable u holds the string for symbol x_{ijk} which is returned along with the string that results from a recursive call which processes the rest of the rule π_{ij}. In case of a token, the symbol is replaced with a string value which is chosen at random from the allowed token values. If x_{ijk} is a non-terminal, a length l and production rule $\pi_{x_{ijk}r}$ is chosen and applied in a recursive call.

```
Algorithm Generate(i, j, k, n))
 1  l ← 1
 2  if x_ijk ∈ Σ
 3     then u ← x_ijk
 4     else  if x_ijk is a token
 5              then u ← random value for x_ijk
 6              else  l ← choose(f'_ijk(n))          ▷ x_ijk is a non-terminal
 7                    r ← choose(f_x_ijk(l))
 8                    u ← Generate(l, x_ijk, r, k + 1)
 9  if k = t_ij
10     then return u
11          return u • Generate(i, j, k + 1, n − l)
```

In this algorithm • represents string concatenation, i.e. $a \bullet b = ab$. The function $choose([l_1, l_2, ..., l_m])$ returns an integer i between 1 and m at random with probability $l_i / \sum_{j=1}^{m} l_j$.

If we assume the first non-terminal is the start symbol, so $N_1 = S$, then the algorithm can be employed by calling $\text{Generate}(1, choose(f_1(n)), 1, n)$.

The generation of a string of length n has a time complexity of $\mathcal{O}(n)$, provided that $f_i(n)$ and $f_{ijk}(n)$ are calculated in a preprocessing step. Calculating tables for f and f' can be done using dynamic programming techniques in $\mathcal{O}(n^2)$ time.

4.1.1 Limitations

The algorithm $\text{Generate}(i, j, k, n))$ has several requirements and limitations pertaining to its usage that are described in this section. It is crucial to have no cycles in the grammar. If there is a cycle, eventually functions f or f' are called repeatedly with the same arguments and termination will be prevented. For non-terminal S, s cycle is a derivation of the following form.

$$S \Rightarrow^* S \tag{4.8}$$

Having no cycles is the only hard requirement for a grammar when it is used with the algorithm. A soft requirement is the absence of empty productions. Empty productions are productions of the form $N \rightarrow \epsilon$. Empty productions can be used since ϵ can be regarded as a symbol from the alphabet, but that will cause it to be counted as having a length of one, which is inconsistent with the concept of the empty string.

There is a fairly straightforward method that removes the empty productions from a grammar in [14], assuming the empty string is not in the language. This method involves rewriting the production rules by replacing all non-terminals S on the right-hand sides for which $S \rightarrow \epsilon$ is in the grammar, by ϵ. This is done until there are no empty productions left in the grammar.

Ambiguous grammars pose a problem for the algorithm as the uniform coverage does not hold for ambiguous strings. In the algorithm, whenever a terminal symbol or string is ambiguous, it will be counted an amount that corresponds to the amount of different leftmost derivations. So this string will also be generated with an increased probability, corresponding to the amount of different derivations. This means if there are two different derivations of a string, the probability of that string being generated gets doubled.

The algorithm *Generate* that we use for string generation uses the string length metric to guarantee termination. Additionally, the length is used such that every string of length n has equal probability of being generated. This uniform coverage based on the length can be undesired in some cases. Example 4.1.1 shows a case where this feature can actually make the probability of generating certain grammatical structures very small.

Example 4.1.1. *Consider the following grammar where S can produce a T or S surrounded with parentheses.*

$$S \rightarrow T \mid (S) \tag{4.9}$$

The symbols (and) have a length of 1. The choice that the algorithm makes between either of these production rules depends on the number of terms of length n that can be made through T versus the sum of the number of terms of length $n-2, n-4, \ldots$ that can be made through T. If the number of terms of length n is far greater than the number of terms of length $n-2$ that can be made through T, the probability of generating through the second production rule is very small.

We encounter this problem in the BES grammar, not only when parentheses are used but also with repetitions. The probability of generating one or two fix-point equations is higher than generating more than two. From a tester's perspective, we want to influence these probabilities such that we can cancel the fact that certain grammatical structures have low probabilities of being generated. Because of the fact that algorithms implemented in tools under test operate on specific parts or grammatical structures of strings from the language, it is desired to be able to increase the probability of generating these grammatical structures.

4.1.2 Weighting

We choose to add the possibility of assigning weights to production rules in the context-free grammar. These weights are then used by the algorithm to make a weighted choice between production rules. So instead of a length based choice the algorithm makes a weighted choice, i.e. the production rule with the highest weight has the highest probability of being chosen by the algorithm to apply to a symbol.

The length is still used. In fact, the function f' is used to choose a length for which a string can be produced and f is used to determine which production rule can produce a string from the chosen length. For a non-terminal symbol N, we can build a list with an entry for each production rule with N as its left-hand side. For rules that cannot build a string of the chosen length, this entry is zero. For all others, this entry is the weight that is assigned to each production rule.

Consider W_π to be the weight of production rule π, then we can modify line 8 of Generate as follows.

$$r \leftarrow choose\left(\left[\left\{\begin{array}{ll} W_{\pi_{ij}} & \text{if } f_{x_{ijk}}[h] > 0 \\ 0 & \text{otherwise} \end{array}\right. \middle| 1 \leq h \leq s_i\right]\right) \tag{4.10}$$

So we have a list of weights and zeros, where function *choose* will return an index for a non-zero entry at random using a weighted selection. The returned index r will gives us production rule $\pi_{x_{ijk}r}$ that we can apply to x_{ijk}.

Weighting is useful in cases where there are several alternatives that are similar to eachother. Consider for example the following three production rules.

$$S \rightarrow E + E \mid E * E \mid E - E \tag{4.11}$$

Using weights, the probability of generating one of the operators can be increased or decreased. In the case of a repetition, weights are less useful.

$$S \rightarrow A \mid AS \tag{4.12}$$

In the above two production rules, weights have little influence on the number of times a recursion is done. Section 7 describes a constraint with which the number of repetitions can be explicitly set.

5 Context

5.1 Meaningful Terms

The algorithm from section 4 generates strings for which we know that they are grammatically correct. However, a randomly generated string can be meaningless or semantically incorrect. Suppose we generate an if-statement where a variable A is used as a guard. The guard of an if-statement has to be a Boolean expression. So this if-statement only makes sense if A is of the Boolean type. If this is not the case, e.g. A is an integer, then the if-statement can be considered meaningless.

```
if A then:
...
else:
...
```

There are several static semantics that can be associated to strings from a language defined by a context-free grammar. In practice some of these semantics can be fulfilled by rewriting the grammar but for many semantic requirements this cannot be done.

In the mCRL2 formalisms there are several semantic requirements that deal with e.g. variable binding and typing. For any randomly generated string, we can only use it as test input if these semantic requirements are valid for this string. Depending on the sematic requirement, the chance of a randomly generated string fulfilling that requirement can be small. In fact, for any randomly generated string of the mCRL2 formalisms, the chance that it is meaningful is very small, making it infeasible to just discard incorrect strings. We need a solution that increases the probability of a single randomly generated string being semantically correct.

We need to add context to the context-free grammars, so that our algorithm is able to determine if the context is valid when making certain decisions in generating a string. Formally, a context-sensitive grammar differs from a context-free grammar in that production rules can have 'context' on the left-hand side.

$$xN_iy \rightarrow x\alpha_{ij}y \tag{5.1}$$

So $N_i \rightarrow \alpha_{ij}$ can be applied if it is in the context defined by x and y. We opt to extend the context-free grammars used in mCRL2 to make them context-sensitive. The two most common context-sensitive grammars are the *two-level grammars* and *attribute grammars*. Norman Paterson [11] has determined that two-level grammars are not suitable for string generation, so we pertain to attribute grammars.

5.2 Attribute Grammars

An attribute grammar is an extension to a context-free grammar where symbols can have several attributes, and the values of those attributes represent a semantic context. Context-sensitive grammars like attribute grammars are widely used in practice in compiler technology as a means to check static semantics during parsing and for usage in code generation [17] [13]. However, we are interested in using attribute grammars to force constraints on the string generator as to only produce strings that are semantically correct. There are several papers that describe such an approach in the form of domain specific languages where production rules and non-terminals of a context-free grammar are enhanced with attributes that will keep track of context-sensitive information during string generation [16] [12].

There are two types of attributes: *inherited* and *synthesized* attributes. Every non-terminal has a set of inherited and synthesized attributes. These sets can be empty, e.g. the start symbol does not have any inherited attributes. The values for inherited attributes are used when a production rule is applied, while values for synthesized attributes are returned after the production rule has been applied.

Every production rule is associated with a set *attribute rules*, that define the values for the attributes per symbol. This means every attribute has a value domain, which can be anything from integers and strings to sets and functions. One attribute rule defines the value for one attribute (inherited or synthesized). Alternatively, an attribute rule can define the string value of a token. This enables us to let the context to determine the value for a token.

Duncan and Hutchinson [2] let attribute values guide production rule selection using *guards*. With guards, we can influence the choice of production rules in the generation algorithm based on the values of the attributes. So if there are multiple choices for a production rule, guards can exclude certain production rules from being chosen. A guard pertaining to production rule π is a Boolean expression over inherited attributes of the left-hand side non-terminal of π. The notion of guards is similar to the notion of conditions of production rules as used in [17] and [19], however conditions are used in parsing to determine if a string is semantically correct, while guards are used to constrain the choice of production rules in the generator.

We can get a formal definition of an attribute grammar from *Compiler Construction* ([19]).

Definition 5.2.1 (Attribute grammar). An attribute grammar is a 4-tuple $(G, A, F, \mathcal{G})$, where

- $G = (V, \Sigma, R, S)$ is a context-free grammar per definition 3.1.1
- every non-terminal $N \in V$ has a possibly empty set of attributes $A(N)$ which consists of two disjoint sets *synthesized* attributes $S(N)$ and *inherited* attributes $I(N)$
  - $A(N) = S(N) \cup I(N)$
  - $I(S) = \emptyset$
- each attribute $a \in A(N)$ has a domain ($dom(a)$) of possible values (integers, strings, sets, functions, etc.)
- each production rule $\pi : X_0 \to X_1X_2...X_n \in R$ has a possibly empty set of *attribute rules*
  - an attribute rule $F_\pi((X_0, 0), a)$ for $a \in S(X_0)$ is a $dom(a)$-typed expression over elements in $I(X_0) \cup A(X_1) \cup A(X_2) \cup ... \cup A(X_n)$
  - an attribute rule $F_\pi((X_i, i), a)$ for $X_i \in V$ and $a \in I(X_i)$ is a $dom(a)$-typed expression over elements in $A(X_0) \cup S(X_1) \cup S(X_2) \cup ... \cup S(X_n) - S(X_i)$
  - for token X_i, an attribute rule $F_\pi((X_i, i))$ is a string valued expression over elements in $A(X_0) \cup S(X_1) \cup S(X_2) \cup ... \cup S(X_n)$
- each production rule $\pi : X_0 \to X_1X_2...X_n \in R$ has a guard $\mathcal{G}_\pi$ which is a Boolean expression over elements in $I(X_0)$

A single attribute rule for production rule π defines the value for a single attribute of a symbol in the left- or right-hand side of π. An attribute rule is an expressions over attributes from other symbols. For tokens, there is a single attribute rule that defines the string value for that token. This is again an expression over attributes from other symbols.

The position of a symbol (the second i in (X_i, i)) is used to distinguish two symbols that are the same non-terminal in a production rule from eachother. They have the same sets of attributes

since they are the same non-terminal, but they can have different values for these attributes at any point. Therefore we combine the non-terminal and its position into an *occurrence tuple*.

To evaluate the expressions of the attribute rules and guards, we can use an environment mapping occurrence tuples to values. So we can formalize a *context*, which contains values for attributes, which we can use to evaluate an attribute rule and subsequently add this value to the context. We define $Z \subseteq (N, \mathbb{N})$ for $N \in V$ as the set containing tuples of non-terminals and their positions. For all production rules $X_0 \to X_1, X_2 ... X_n$, $(X_i, i) \in Z$ if X_i is a non-terminal. Furthermore, we define $A = A(N)$ for $N \in V$ as the set containing all attributes and $\zeta = \{dom(a) \mid a \in A\}$ as the set of all attribute domains.

Definition 5.2.2 (Context). Let the context $\mathcal{C} : Z \times A \to \zeta$ be an environment mapping symbol/attribute tuples to a value from the attribute's domain, where

- $\forall (X, i) \in Z, a \in I(X) \cup S(X) : \mathcal{C}((X, i), a) \in dom(a)$
- for attribute rule $F_\pi((X, i), a)$, and context $\mathcal{C}$, the evaluation $F_\pi((X, i), a)$ with respect to context $\mathcal{C}$, is denoted $[\![F_\pi((X, i)a)]\!]\mathcal{C}$ and involves calculating the value for expression $F_\pi((X, i), a)$ by substituting its variables by their values from $\mathcal{C}$
  - $[\![F_\pi((X, i), a)]\!]\mathcal{C} \in dom(a)$

For the evaluation of guards we can use a context as well. So we have a guard $\mathcal{G}_\pi$ of production rule $\pi : X_0 \to X_1 X_2 ... X_n$ and context $\mathcal{C}$ for which it holds that $\mathcal{C}((X_0, 0), a)$ is defined for all $a \in I(X_0)$. Then $[\![\mathcal{G}_\pi]\!]\mathcal{C}$ evaluates to true or false.

Now we can design an algorithm that builds a string while evaluating attribute rules and guards and choosing production rules for which the guard is true. We can start with an algorithm that applies a production rule, after which we add length and length control back while dealing with termination.

The ApplyProductionRule algorithm listed below takes a production rule π and context $\mathcal{C}$ and returns a string u and context $\mathcal{C}'$. The algorithm processes the non-terminals in the right-hand side of the given production rule and recursively applies the algorithm to produce a terminal string. For every non-terminal X_i, first its inherited attribute values are calculated and put into context $\mathcal{C}'$. Then a production rule is chosen based on the evaluation of the guard, after which a recursive call is made which will result in a terminal string w and new context $\mathcal{C}''$. Context $\mathcal{C}$ is updated with values of the synthesized attributes of X_i from $\mathcal{C}''$. Tokens are assigned their string values through their attribute rules and finally u, now being a terminal string, is returned as well as context $\mathcal{C}'$ containing values for the synthesized attributes of X_0, the left-hand side non-terminal.

If S is the start symbol of our attribute grammar, we can apply the algorithm by choosing a production rule $\pi : S \to \alpha$ and do $(u, \mathcal{C}) \leftarrow \text{ApplyProductionRule}(\pi, \emptyset)$. Now u is our generated terminal string and $\mathcal{C}$ is the resulting context. Since the resulting context is not relevant we can discard it.

```
ApplyProductionRule(π : X_0 → X_1 X_2 ... X_n, C)
 1  u ← X_1 X_2 ... X_n
 2  while there are non-terminals in u
 3        do take non-terminal X_i
 4           C' ← ∅
 5           for a ∈ I(X_i)
 6                do C'((X_i, i), a) ← ⟦F_π((X_i, i), a)⟧C
 7           choose new production rule π' with left-hand side X_i for which ⟦G_π'⟧C' is true
 8           (w, C'') ← ApplyProductionRule(π', C')
 9           replace X_i by w in u
10           for a ∈ S(X_i)
11                do C((X_i, i), a) ← C''((X_i, i), a)
12  for tokens X_i in u
13        do replace X_i with ⟦F_π((X_i, i))⟧C in u
14  C' ← ∅
15  for a ∈ S(X_0)
16        do C'((X_0, 0), a) ← ⟦F_π((X_0, 0), a)⟧C
17  return (u, C')
```

Lines 2 and 3 of the algorithm imply an arbitrary order in which the non-terminals are processed. However, the order of processing or *traversal order* is important due to the dependence of attributes. The attribute rules for a symbol are expressions over attributes from other symbols, so an attribute can depend on several other attributes. Any traversal of symbols in a production rule should respect these dependencies in order to prevent undefined behaviour. Section 5.4 describes several traversal strategies and the cases in which they are applicable.

The following example shows an attribute grammar that defines a language that cannot be defined using a context-free grammar. The example illustrates how attributes can be used for context and how a string is generated using the algorithm.

Example 5.2.1. *Consider the following attribute grammar that defines the non-context-free language* $\{a^n b^n c^n \mid n \geq 1\}$.

Production rules		*Attribute rules*	*Guards*
$\pi_{11}:$	$S \rightarrow ABC$	$B.n_I \leftarrow A.n_S$ $C.n_I \leftarrow A.n_S$	
$\pi_{21}:$	$A_0 \rightarrow a$	$A_0.n_S \leftarrow 1$	
$\pi_{22}:$	$A_0 \rightarrow aA_1$	$A_0.n_S \leftarrow A_1.n_S + 1$	
$\pi_{31}:$	$B_0 \rightarrow b$		$B_0.n_I = 1$
$\pi_{32}:$	$B_0 \rightarrow bB_1$	$B_1.n_I \leftarrow B_0.n_I - 1$	$B_0.n_I > 1$
$\pi_{41}:$	$C_0 \rightarrow c$		$C_0.n_I = 1$
$\pi_{42}:$	$C_0 \rightarrow cC_1$	$C_1.n_I \leftarrow C_0.n_I - 1$	$C_0.n_I > 1$

Here, n *is an attribute that holds the number of occurrences. The attribute* n *is synthesized for non-terminal* A*, denoted* n_S *and inherited for symbols* B *and* C*, denoted* n_I*.*

Starting with symbol S*, the first production rule is applied giving us the sentential form* ABC*. The symbols are processed from left to right and for* A *there are no inherited attributes to evaluate attribute rules for. If* π_{22} *and subsequently* π_{21} *are applied we have the following sentential form and context.*

term	*context*
$aaBC$	$((A.n_S, 1), 2)$

Here, $(A.n_S, 1)$ *is the attribute of* A *with the position* 1 *of* A *in the production rule and the value of the attribute is* 2*. Now for* B*, the algorithm assigns the value for inherited attribute* n_I *to the value of* $A.n_S$*. The guards will limit the choice of production rules to* π_{32}*. Rule* π_{32} *is applied once, producing* bB*. Now* $B.n_I = 1$*, so* π_{31} *is applied and we get the following sentential form and context.*

term	*context*
$aabbC$	$((A.n, 1), 2)$ $((B.n, 2), 2)$

Now we have produced two as and two bs. For non-terminal C*, the same is done as with* B*, which results in the following terminal string and context.*

term	*context*
$aabbcc$	$((A.n_S, 1), 2)$ $((B.n_I, 2), 2)$ $((C.n_I, 3), 2)$

Of course, the context is irrelevant after the final result is returned. However, this shows how the context is updated after applications of production rules.

5.3 Length

Until now we have abstracted from the length control used in the string generation algorithm for context-free grammars to enforce termination. Since we still want termination we can add the length control back. We can combine GENERATE from section 4 and APPLYPRODUCTIONRULE from section 5.2 into a single algorithm. This algorithm generates a string from an attribute grammar, while imposing length control.

Recall the index notation used in section 4 to describe the elements of a grammar. For example, we can denote the production rules as follows.

$$R = \{\pi_{ij} : N_i \rightarrow x_{ij1}x_{ij2}...x_{ijt_{ij}} \mid 1 \leq i \leq r \wedge 1 \leq j \leq s_i\} \tag{5.2}$$

Where r is the amount of non-terminals, s_i is the amount of production rules with left-hand side N_i and t_{ij} is the amount of symbols in the right-hand side of π_{ij}. Due to the recursive nature of GENERATE, the resulting algorithm employs the left-right strategy by iterating k from 1 to t_{ij}, which effectively means iterating through the symbols of the right-hand side of a production rule from left to right.

Listed below is the resulting algorithm. For production rule $\pi_{ij} : N_i \rightarrow x_{ij1}x_{ij2}...x_{ijk}$, GENERATEII$(i, j, k, n, \mathcal{C})$ generates a string starting at symbol x_{ijk} using context $\mathcal{C}$.

```
GENERATEII(i, j, k, n, C)
 1  l ← 1
 2  if x_ijk ∈ Σ
 3     then u ← x_ijk
 4     else if x_ijk is a token
 5             then u ← [[F_πij((x_ijk, k))]]C
 6             else C' ← ∅
 7                  for all a ∈ I(N_x_ijk)
 8                       do C'((x_ijk, k), a) ← [[F_πij((x_ijk, k), a)]]C
 9                  l ← choose(f'_ijk(n))
10                  r ← choose(h(x_ijk, l, C'))
11                  (u, C'') ← GENERATEII(x_ijk, r, 1, l, C')
12                  for a ∈ S(N_x_ijk)
13                       do C((x_ijk, k), a) ← C''((x_ijk, k), a)
14  if k = t_ij
15     then C' ← ∅
16          for all a ∈ S(N_i)
17               do C'((N_i, 0), a) ← [[F_πij((N_i, 0), a)]]C
18          return (u, C'')
19  (w, C'') ← GENERATEII(i, j, k + 1, n − l, C)
20  return (u • w, C'')
```

The choice for r is based on function h that returns a list of weights for rules $\pi_{x_{ijk}r}$, with $1 \leq r \leq s_i$ if the guard for that rule evaluates to true and it can build a string of length m and zero otherwise. So in line 10, a weighted selection is made between all the rules that have valid lengths and guards.

$$h(i, n, \mathcal{C}) = \left[\left\{ \begin{array}{ll} W_{ij} & \text{if } [\![\mathcal{G}_{\pi_{ij}}]\!]\mathcal{C} \wedge sum(f'_{ij1}(m)) > 0 \\ 0 & \text{otherwise} \end{array} \right. \middle| \; 1 \leq j \leq s_i \right]$$

After application of $\pi_{x_{ijk}r}$ we have a substring u so we can add u to the resulting string and continue processing for $k+1$. Finally we calculate values for the synthesized attributes of N_i and

return them along with string $u \bullet w$. Basically u is the string representing symbol x_{ijk} and w is the string representing the rest of the production rule.

From example 5.2.1 an issue arises concerning the lengths that will be chosen by the algorithm. The choice for the lengths of substrings does not take the context into account. If a certain length is chosen for A when π_{11} is applied, this indirectly forces B and C to be of the same length, regardless of the choice the algorithm makes. So there is a conflict on the expected length and the length that is allowed due to the constraints.

Production rule guards are used to exclude certain production rules from being chosen to be applied by the algorithm, when the current value for the attributes (context) violates a constraint imposed on the generation for that production rule. It can be the case that these exclusions make sure there are no choices to be made anymore. In other words, the string generation algorithm can block because it cannot generate a string of length n, since the guards disallow any production rule that can make a string of length n.

This example of instantiation of propositional variables shows a fundamental problem when the algorithm is not allowed apply any production rule due to the context and length. A solution is to stop generation and discard the intermediate result and try again. However, the probability of generating a string successfully can be very low, so this is not feasible.

We make an assumption that for every possible context there is a production rule for which the guard evaluates to true. This means that there is always an alternative production rule to choose from. The problem now arises when the only valid choices cannot generate a string of the specified length. For specified length n, and chosen production rule π_{ij} for which the guard is true, but $f'_{ij1}(n) = 0$, we have to find new length m, such that $f'_{ij1}(m) > 0$. We apply the heuristic where $|n - m|$ is minimal for all m where $f'_{ij1}(m) > 0$.

5.3.1 Termination

Theoretically it is possible for ApplyProductionRule and GenerateII to not terminate, for example in the case where for a symbol S and a context, the guard every production rule with left-hand side S of every production rule evaluates to false. Furthermore, there can be a cycle where the choice for length m leads to the same situation with the same length and guard problem. Consider for example the following attribute grammar.

Example 5.3.1. *Consider the following attribute grammar.*

$$\begin{array}{ll|c|c} \multicolumn{2}{c|}{\text{Production rules}} & \text{Attribute rules} & \text{Guards} \\ \pi_{11}: & A \to B & I(B)[E] \leftarrow random([0..1]) & \\ \pi_{21}: & B \to aA & & \\ \pi_{22}: & B \to b & & I[E] > 0.5 \end{array} \tag{5.3}$$

Function random chooses a real valued number uniformly at random from the supplied interval. If we generate a string of length 1*, a real number between 0 and 1 will be assigned to attribute E of* $I(B)$*. The guard is only true for a value for* $E > 0.5$*. This means every time a production rule for B is applied, the algorithm has a* 0.5 *probability of terminating.*

Consequently, one can design an attribute grammar that never terminates: it runs into a cycle where the heuristic of choosing a length has to applied continuously.

Since we choose new lengths in order to satisfy certain language requirements and make sure the algorithm makes progress, we know that the resulting length differs from given length n.

Basically, the semantic requirements do not allow for certain length substrings. The algorithm makes new choices for lengths such that it can generate strings while still maintaining the static semantics. This means that the resulting length can differ from the given length.

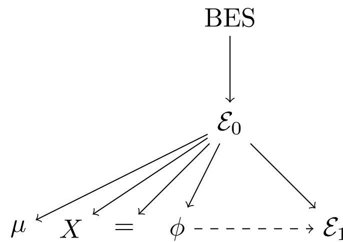
To summarize, termination with GenerateII is only guaranteed whenever there is always a production rule to choose from, i.e. for every symbol S and context there is a production rule with left-hand side S for which the guard evaluates to true. Furthermore we assume the absence of cycles where the heuristic of choosing a new length is continuously applied.

5.4 Traversal Strategies

When attribute grammars are used in compilers or in our case for string generation, there has to be a strategy that determines the order in which non-terminals are processed. The reason is that the attributes of a symbol can depend on another symbol's attributes. This dependency has to be respected since we cannot evaluate an attribute rule if the values for the variables in that rule are unknown. This dependency does not only occur between symbols of a single production rule, but can occur between symbols from different production rules as well.

$$BES \Rightarrow^* (\mu X = \phi)\mathcal{E} \tag{5.4}$$

Consider the building of a BES where we encounter the above sentential form. We want to allow the usage of propositional variables inside ϕ, that will be bound as the left-hand side of fix-point equations in $\mathcal{E}$. Assume we have a synthesized environment attribute for $\mathcal{E}$ that holds the list of bound variables, then ϕ depends on this attribute. The following (incomplete) derivation tree visualizes this dependency in a concrete syntax tree.



Compiler technology distinguishes various subsets of attribute grammars that pertain to the traversal strategy. *Well-defined* attribute grammars ([7]) are a subset of attribute grammars where there are no circular dependencies. Commonly in top-down parsing, *l-attributed* ([6]) grammars are used. In l-attributed grammars, attributes only depend on other attributes from symbols occurring to the left. This means everything can be processed in one left-to-right traversal.

The attribute grammars used in the toolset are in the class of *ordered attribute grammars* [6], a subset of well-defined attribute grammars, where for each symbol a partial order over the associated attributes can be defined, such that in any context there is a traversal strategy that respects this order.

The algorithm ApplyProductionRule does support any order of attribute dependencies between symbols of the same rule. However, dependencies between different rules are not supported. For the algorithm GenerateII there is even less freedom as only attributes from symbols to the left can be used, since symbols are processed from left-to-right. In the implementation (section 9) we will allow the postponing of processing symbols whose attributes depend on symbols that are generated at a later point, until these symbols are generated. This second pass is necessary to support several static semantics like variable binding.

For the attribute grammars implemented in the testing framework these requirements hold, so the algorithm terminates when used with these attribute grammars.

6 Constraints

The semantic requirements for strings we generate are modelled using an attribute grammar. We can model a single semantic requirement by enhancing a context-free grammar with attributes, attribute rules and guards. The idea is that every string from the resulting attribute grammar adheres to this semantic requirement. We introduce *constraints*, which are attribute grammars that aim to define only strings adhering to certain semantic requirements.

Definition 6.0.1 (Constraint)**.** A *constraint* on a context-free grammar G, with respect to a semantic requirement is an attribute grammar $AG = (G, A, F, \mathcal{G})$ such that this semantic requirement is valid for all $\sigma \in L(AG)$.

Multiple constraints can be combined into a single attribute grammar by adding together the attributes and attribute rules and taking the conjunction of the guards of every production rule. Combining attribute grammars in this way can be done when the base context-free grammar is the same for the attribute grammars.

This section describes the semantic requirements that have to be fulfilled and for every language requirement, we present a constraint that solves this requirement. Because a formal definition of a semantic requirement is often done using structural definition on a grammar, we use the attribute grammars to explain the semantic requirements as well.

6.1 Variable Binding

Variable binding or declaration is an important aspect in the formalisms in mCRL2. There are several types of variables like data variables, propositional variables and process variables. Commonly, tools from the mCRL2 toolset only accept input terms where every variable is bound before it is used. So bound-before-usage is a semantic requirement to which our generated strings must adhere. Consider for example the following BES.

$$(\mu X = X \wedge Y) \tag{6.1}$$

In this BES, the propositional variable X is bound through the left-hand side of the fix-point equation. The propositional variable Y is free, it is not bound in any fix-point equation in this BES. This violates our requirement and we want to prevent our string generation algorithm from generating such a BES.

For any BES we generate, it is required that every propositional variable is bound before it is used (no free variables occur, also called *closed*) and every propositional variable is bound at most once. We can formalize this as follows. We can define $occ(\mathcal{E})$, for a BES $\mathcal{E}$ as the set of variables occurring on the right-hand side of the fix-point equations in $\mathcal{E}$. We can define $bnd(\mathcal{E})$ as the set of variables occurring on the left-hand side of the fix-point equations in $\mathcal{E}$ and let $len(\mathcal{E})$ be the amount of fix-point equations in $\mathcal{E}$. Now the requirements can be formalized as $occ(\mathcal{E}) \subseteq bnd(\mathcal{E})$ and $\#bnd(\mathcal{E}) = len(\mathcal{E})$.

To adhere to these requirements, we have to know for every usage of a variable, if it is allowed by checking what has already been generated. This means we have to be able to check whether or not it has been declared before and possibly if it has the correct type. To do this, we add an inherited and synthesized environment attribute Env_I and Env_S to every non-terminal in the grammar except for the start symbol. The value domain of this attribute is sets of variables for every type of variable (action, propositional-, data- or process variable, etc.). So for the PBES grammar, Env consists of a set of data variables and a set of propositional variables, i.e. $Env_I = Env_I^{prop} \cup Env_I^{data}$.

A variable in one of the sets of Env has a name and possibly a subtype (in the case of a data variable). Subtypes are integer, natural number, Boolean, etc., but also more complex types are possible. We add a variable to Env when we generate a declaration and we draw from Env when we generate a variable usage.

Production rule	*Attribute rule*
$BES \rightarrow \mathcal{E}$	$\mathcal{E}.Env_I^{prop} \leftarrow \emptyset$

In the first production rule of the BES grammar, with the start symbol on the left-hand side, we initialize the set of propositional variables Env_I^{prop} to the empty set.

Production rule	*Attribute rules*
	$X \leftarrow choice(\mathcal{X} - \mathcal{E}.Env_I^{prop})$
$\mathcal{E} \rightarrow (\sigma X = \phi)$	$\phi.Env_I^{prop} \leftarrow \mathcal{E}.Env_I^{prop} \cup \{X\}$
	$\mathcal{E}.Env_S^{prop} \leftarrow \phi.Env_S^{prop}$

When we generate a fix-point equation we also bind a propositional variable. This means that we can add this variable to our environment. To generate a unique variable name, we take the set of allowed variable names $\mathcal{X}$ and subtract those elements already in our environment and we choose an element uniformly at random from the resulting set through the external function *choice*. So we assign a unique variable name to token X. We can then add X to Env_I^{prop} inherited by ϕ and synthesized by $\mathcal{E}$ in the second and third attribute rule. After we are done generating this fix-point equation, the resulting environment will contain X as a bound propositional variable.

Production rule	*Attribute rule*
$\phi \rightarrow X$	$X \leftarrow choice(\phi.Env_I^{prop})$
	$\phi.Env_S^{prop} \leftarrow \phi.Env_I^{prop}$

In a BES, a variable usage is generated through production rule $\phi \rightarrow X$. We can make a selection uniformly at random, from the inherited attribute value $\phi.Env_I^{prop}$. We assign the selected variable name to token X which is subsequently generated in the resulting string. By doing this we make sure that every usage of a variable occurs within a context where it is bound by being the left-hand side of a fix-point equation.

For formalisms other than BESs, where other types of variables can occur, the constraints are modeled similar to the constraints for the propositional variable case. For every type of variable there are production rules that bind these variables and production rules that use these variables. So for every type of variable we use a set to which we add a variable when we generate a binding and from which we draw when we generate a usage.

Variable binding is a case where second pass is required. The reason is that the scoping rules of mCRL2 formalisms allow for *forward references* of, for example, propositional variables, as described in section 5.4. We mark the production rules producing an instance of a propositional variable so that when the algorithm tries to apply this production rule, it will postpone the application until all production rules that bind variables have been processed.

6.2 Monotonicity

Monotonicity is a property of the right-hand side of a fix-point equation in BESs and PBESs. A Boolean expression ϕ is monotonous when it can be rewritten such that a propositional variable is not inside the scope of a negation. Every BES and PBES we generate is required to be monotonous. When we take the BES grammar with operator precedence, we can formally define monotonicity as

follows. Note that without operator precedence this definition is not possible, due to ambiguities occurring in the grammar.

Definition 6.2.1 (Monotonicity)**.** A BES $\mathcal{E}$ is *monotonous*, if for the right-hand side ϕ of every fix point equation in $\mathcal{E}$, $m(\phi)$ is true, with m defined as follows.

$$\begin{array}{rcl}
m(\neg b) & =_{def} & true \\
m(\neg\neg\phi) & =_{def} & m(\phi) \\
m(\neg(\phi_0 \Rightarrow \phi_1)) & =_{def} & m(\phi_0) \wedge m(\neg\phi_1) \\
m(\neg(\phi_0 \wedge \phi_1)) & =_{def} & m(\neg\phi_0) \wedge m(\neg\phi_1) \\
m(\neg(\phi_0 \vee \phi_1)) & =_{def} & m(\neg\phi_0) \wedge m(\neg\phi_1) \\
m(\neg X) & =_{def} & false \\
m(b) & =_{def} & true \\
m(\phi_0 \Rightarrow \phi_1) & =_{def} & m(\neg\phi_0) \wedge m(\phi_1) \\
m(\phi_0 \wedge \phi_1) & =_{def} & m(\phi_0) \wedge m(\phi_1) \\
m(\phi_0 \vee \phi_1) & =_{def} & m(\phi_0) \wedge m(\phi_1) \\
m(X) & =_{def} & true
\end{array} \quad (6.2)$$

Two operators introduce negation, namely implication ($\Rightarrow$) and negation itself ($\neg$). For implication, negation is applied to the left operand. We can see from the definition that in all other cases the monotonicity directly depends on the subexpressions.

We model the semantic requirement monotonicity into an attribute grammar, by adding a single inherited attribute m to the non-terminal ϕ in the BES grammar. In the case of PBESs this approach is analogous. The value for m is true if the current scope is monotonous and false otherwise. When an occurrence of a propositional variable is generated and the inherited value for m is false, we have to negate this propositional variable.

Production rule	*Attribute rule*
$\phi \to X$	$X \leftarrow \begin{cases} choice(\phi.Env_I^{prop}) & \text{if } \phi.m \\ \neg \bullet choice(\phi.Env_I^{prop}) & \text{if } \neg\phi.m \end{cases}$

Basically if we want to insert a propositional variable under the scope of negation, we add an extra negation such that it does not occur under the scope of negation anymore.

For rules other than implication and negation, we just pass down the inherited value for m. For example for the conjunction production rule we have the following attribute rules.

Production rule	*Attribute rules*
$\phi_0 \to \phi_1 \wedge \phi_2$	$\phi_1.m \leftarrow \phi_0.m$ $\phi_2.m \leftarrow \phi_0.m$

The only modifications done to the value of m are done with implication and negation. For negation, we can invert the scope as follows.

Production rule	*Attribute rule*
$\phi_0 \to \neg\phi_1$	$\phi_1.m \leftarrow \neg(\phi_0.m)$

For implication, the scope of the left operand is inverted, while for the right operand it is kept the same.

Production rule	*Attribute rules*
$\phi_0 \to \phi_1 \Rightarrow \phi_2$	$\phi_1.m \leftarrow \neg(\phi_0.m)$ $\phi_2.m \leftarrow \phi_0.m$

These are the rules needed to enforce monotonicity. Any BES generated using this attribute grammar is monotonous. As stated before, in the case of PBESs the constraint is analogous.

6.3 Operator Precedence

Recall from section 3.1.1 that we require operator precedence to resolve ambiguities in the grammars we use. Here we describe a way to employ operator precedence in the form of a constraint. We show the constraint on a BES, but the solution is generic, it can be used with any set of production rules for which assigning priorities resolves ambiguities.

We assign an inherited attribute P to any non-terminal N that produces an operation and for which we need to resolve the ambiguities. For each production rule π with left-hand side N, we define a guard $N.P \leq p$, where $p \geq 1$ is the priority we want for π. The highest priority operator has the lowest p value, so the guard for the highest priority operation is $N.P \leq 1$.

For a binary operation, if it is left associative, we increase the value for P of the right operand by 1, while keeping the value for P for the left operand at the current priority level. For right associativity this is the other way around. The current priority level is in fact the value for p we have assigned to the production rule. This can differ from the inherited value of P.

In a BES, we have the non-terminal ϕ for Boolean expressions that requires operator precedence. The highest priority operation is implication ($\Rightarrow$). We can define the guard and attribute rules for implication as follows.

Production rule	*Attribute rules*	*Guard*
$\phi_0 \rightarrow \phi_1 \Rightarrow \phi_2$	$\phi_1.P \leftarrow 2$ $\phi_2.P \leftarrow 1$	$\phi_0.P \leq 1$

Since $\Rightarrow$ is the highest priority operation, we assign the lowest value for p, namely 1, to it. Because it is right associative, the left operand is set to lower priority (higher p value). This means for example that for the right operand another implication cannot be generated directly.

We omit the examples for the other operators since they are analogous, except for the parentheses. When parentheses are generated, we can reset the priority level, since parentheses are inherently disambiguating.

Production rule	*Attribute rules*	*Guard*
$\phi_0 \rightarrow (\phi_1)$	$\phi_1.P \leftarrow 1$ $\phi_2.P \leftarrow 1$	$\phi_0.P \leq 4$

The priority level for parentheses is 4, since in the BES grammar there are three operations with higher precedence (implication, conjunction and disjunction). For the fix-point generating production rule we can initialize the priority level to the lowest value.

Production rule	*Attribute rules*
$\mathcal{E} \rightarrow (\sigma X = \phi)$	$\phi.P \leftarrow 1$

This shows how operator precedence can be explicitly used in an attribute grammar.

6.4 Typing

Variable typing is a common occurrence in programming languages. Often, correct typing is a semantic requirement for a program for it to able to execute. Several mCRL2 formalisms contain

the notion of data: variables that have a data type, which can be user-defined or one of the built-in types integer ($\mathbb{Z}$), rational ($\mathbb{R}$), natural number ($\mathbb{N}$), positive number ($\mathbb{P}$) or Boolean ($\mathbb{B}$). The use of data in the mCRL2 formalisms has to be such that the typing is correct. When a conjunction occurs between two data variables, they have to be of the Boolean type.

We can model the semantic requirement of typing into a constraint. We start by adding an inherited and synthesized attribute $Type_I$ and $Type_S$ to the non-terminal producing data expressions. Every operation on data has a set of expected types for each operand and a resulting type. The expected types are inherited, while the resulting type is synthesized. We add guards to the operations such that we prevent operations where the resulting type differs from the expected type.

The value domain for $Type_I$ and $Type_S$ is $\mathcal{P}(\mathcal{T})$ where $\mathcal{T}$ is the set of all types and $\mathcal{P}$ is the powerset operator. The expected type can be can be one of many; several operators can work on different types. The resulting type is usually a singleton.

We can get the semantic requirement from the documentation of mCRL2 where each operation is defined. For the Boolean type, we have the following type definitions.

$$\begin{array}{ccc} \textit{Operation} & \textit{Type} & \textit{Semantics} \\ \neg\psi & \mathbb{B} \to \mathbb{B} & \text{Negation} \\ \psi_0 \wedge \psi_1 & \mathbb{B} \times \mathbb{B} \to \mathbb{B} & \text{Conjunction} \\ \psi_0 \vee \psi_1 & \mathbb{B} \times \mathbb{B} \to \mathbb{B} & \text{Disjunction} \\ \psi_0 \Rightarrow \psi_1 & \mathbb{B} \times \mathbb{B} \to \mathbb{B} & \text{Implication} \\ \forall \mathcal{D}.\psi & \mathbb{B} \to \mathbb{B} & \text{Universal quantification} \\ \exists \mathcal{D}.\psi & \mathbb{B} \to \mathbb{B} & \text{Existential quantification} \end{array} \tag{6.3}$$

Where $\mathcal{D}$ is the non-terminal that produces a binding of a sequence of one or more data variables. Through these type definitions, we know the type of the operands and the resulting type. The production rule for the conjunction operator gets the following guard and attribute rules.

Production rule	*Attribute rules*	*Guard*
$\psi_0 \to \psi_1 \wedge \psi_2$	$\psi_1.Type_I \leftarrow \{\mathbb{B}\}$ $\psi_2.Type_I \leftarrow \{\mathbb{B}\}$ $\psi_0.Type_S \leftarrow \{\mathbb{B}\}$	$\mathbb{B} \in \psi_0.Type_I$

Here, ψ_1 and ψ_2 get the Boolean type assigned to their inherited attributes. Note that the rule $\psi_0.Type_S \leftarrow \mathbb{B}$ defines the value for the *synthesized* attribute $Type_S$ while the guard uses the value for the inherited attribute $Type_I$. The guard prevents this production rule from being applied, when the Boolean type is not one of the expected types.

Comparison operations can work on any type for which two values can be compared. We have the following type definition for the comparison operators, with S being any type.

$$\begin{array}{ccc} \textit{Operation} & \textit{Type} & \textit{Semantics} \\ \psi_0 = \psi_1 & S \times S \to \mathbb{B} & \text{Equality} \\ \psi_0 \neq \psi_1 & S \times S \to \mathbb{B} & \text{Inequality} \\ \psi_0 < \psi_1 & S \times S \to \mathbb{B} & \text{Less than} \\ \psi_0 > \psi_1 & S \times S \to \mathbb{B} & \text{Greater than} \\ \psi_0 \leq \psi_1 & S \times S \to \mathbb{B} & \text{Less than or equal to} \\ \psi_0 \geq \psi_1 & S \times S \to \mathbb{B} & \text{Greater than or equal to} \end{array} \tag{6.4}$$

We can show the attributes and guard for the equality operation. For the rest of the operations this is analogous, since they all have the same typing definition.

Production rule	*Attribute rules*	*Guard*
$\psi_0 \rightarrow \psi_1 = \psi_2$	$\psi_1.Type_I \leftarrow \mathcal{T}$ $\psi_2.Type_I \leftarrow \psi_1.Type_S$ $\psi_0.Type_S \leftarrow \{\mathbb{B}\}$	$\mathbb{B} \in \psi_0.Type_I$

The guard results from the type definition: the equality operator is of the Boolean type. From the type definition we know that the operands can be any type ($\mathcal{T}$), but the left operand is required to be of the same type as the right operand. This means we can use the value for $Type_S$ for the left operand as the value for $Type_I$ for the right operand. The choice for the left operand is made due to the left-right traversal strategy that is employed. We first need to generate the left operand to know the type of the right operand.

For numeric types there are a multitude of operations with more complex type definitions. Listed below is the type definition for the addition operator and it shows that the resulting type depends on the type of the operands. Also, for the type $\mathbb{P}$ (positive number) the left and right operand is allowed to differ in type.

$$\begin{array}{ccc} \textit{Operation} & \textit{Type} & \textit{Semantics} \\ & \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P} & \\ & \mathbb{P} \times \mathbb{N} \rightarrow \mathbb{P} & \\ \psi_0 + \psi_1 & \mathbb{N} \times \mathbb{P} \rightarrow \mathbb{P} & \text{Addition} \\ & \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} & \\ & \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} & \\ & \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} & \end{array} \tag{6.5}$$

In our attribute rules we have to take into account the different attribute types possible per operand. Again, since we employ the left-right strategy, we let the left operand determine the type of the right operand.

$$\begin{array}{c|l|c} \textit{Production rule} & \textit{Attribute rules} & \textit{Guard} \\ & \psi_1.Type_I \leftarrow \{\mathbb{P}, \mathbb{N}, \mathbb{Z}, \mathbb{R}\} & \\ & \psi_2.Type_I \leftarrow \begin{cases} \{\mathbb{P}, \mathbb{N}\} & \text{if } \mathbb{P} \in \psi_1.Type_S \\ \{\mathbb{P}, \mathbb{N}\} & \text{if } \mathbb{N} \in \psi_1.Type_S \\ \psi_1.Type_S & \text{otherwise} \end{cases} & \\ \psi_0 \rightarrow \psi_1 = \psi_2 & & \{\mathbb{P}, \mathbb{N}, \mathbb{Z}, \mathbb{R}\} \cap \psi_0.Type_I \neq \emptyset \\ & \psi_0.Type_S \leftarrow \begin{cases} \{\mathbb{P}\} & \text{if } \mathbb{P} \in \psi_1.Type_S \\ \psi_2.Type_S & \text{otherwise} \end{cases} & \end{array}$$

These examples show that there is a straightforward correspondence between the type definition and the attribute rules and guards. Therefore we omit the examples for the other operations.

6.5 Guarded Processes

This constraint is only applicable to mCRL2 process specifications. The context-free grammar for mCRL2 process specifications is listed in appendix A. In mCRL2 process specifications every process is required to be guarded. Guarded means that there is no left recursion for processes expressions. Formally we structurally define the notion of guardedness with a definition for the

Boolean function $g(p)$ for process expressions as follows. In this definition and constraint we abstract from data. Actions and processes can contain data parameters, but they have no influence on the guardedness of a process.

Definition 6.5.1 (Guardedness)**.** A process expression p is *guarded* if $g(p)$ is true, with g defined as follows.

$$\begin{array}{rcl} g(p) & =_{def} & g(p, \emptyset) \\ g(\alpha, V) & =_{def} & true \\ g(\delta, V) & =_{def} & true \\ g(\tau, V) & =_{def} & true \\ g(P, V) & =_{def} & \begin{cases} false & \text{if } P \in V \\ g(p, V \cup \{P\}) & \text{if } P \notin V \end{cases} \\ & & \text{where } p \text{ is the equation for } P \\ g(p_0 + p_1, V) & =_{def} & g(p_0, V) \wedge g(p_1, V) \\ g(p_0 \cdot p_1, V) & =_{def} & g(p_0, V) \\ g(c \to p, V) & =_{def} & g(p, V) \end{array} \tag{6.6}$$

We add synthesized attribute G_S and inherited attributes G_I and P_I to the non-terminals in the grammar except for the start symbol. The attributes G_S and G_I keep track of the processes that are unguarded in the current scope. The attribute P_I represents the name of the process that is being bound by the current expression. When a process variable is used, it has to be in the scope where it is guarded. If this is not the case, we choose to prefix the process variable with an action. Actions are stored in the Env_I^{act} attribute.

Production rule	*Attribute rules*
$p \to P$	$P \leftarrow \begin{cases} P' & \text{if } P' \notin G_I \\ choice(p.Env_I^{act}) \bullet P' & \text{if } P' \in G_I \end{cases}$ where $P' = choice(p.Env_I^{proc})$

So we choose a process variable P', after which we consult the set of unguarded processes. If $P' \in G_I$, we have to guard it with an action which is chosen randomly from the set $p.Env_I^{act}$. This shows how the usage of a process variable fulfills the guardedness requirement. The grammar is such that there is always at least one process and one action, so $p.Env_I^{proc}$ and Env_I^{act} are not empty.

An action guards the current process $p.P_I$, so we can remove it from the set of unguarded processes.

Production rule	*Attribute rules*
$p \to \alpha$	$\alpha \leftarrow p.Env_I^{act}$ $p.G_S \leftarrow p.G_I - \{p.P_I\}$

This is analogous for the actions δ and τ. When a sequence $(\cdot)$ is generated, the current process can become unguarded in the left operand. Therefore we use the set of unguarded processes of the left operand for the right operand. Likewise, the current process can become unguarded in the right operand, so we return the final set of unguarded processes through the right operand.

Production rule	*Attribute rules*
$p_0 \to p_1 \cdot p_2$	$p_1.G_I \leftarrow p_0.G_I$ $p_2.G_I \leftarrow p_1.G_S$ $p_0.G_S \leftarrow p_2.G_S$

With the choice operator (+), anything that becomes guarded in p_1 is still unguarded in p_2. So both operands inherit the set of unguarded processes from p_0. The union for $p_0.G_S$ is done because when a process becomes guarded in p_1 it is still unguarded in p_2 as well as for the complete process expression. Only if in both subexpressions the same process becomes guarded, it will not exist in the resulting value for $p_0.G_S$.

Production rule	*Attribute rules*
$p_0 \to p_1 + p_2$	$p_1.G_I \leftarrow p_0.G_I$ $p_2.G_I \leftarrow p_0.G_I$ $p_0.G_S \leftarrow p_1.G_S \cup p_2.G_S$

The attribute rules for the other production rules of p can be determined through the definition of guardedness, similar to the above examples. For the production rule that binds a process variable, we can consider the following attribute rules.

Production rule	*Attribute rules*
$\mathcal{E} \to P = p$	$P \leftarrow choice(\mathcal{P} - \mathcal{E}.Env_I^{proc})$ $p.G_I \leftarrow \mathcal{E}.G_I \cup \{P\}$ $p.P_I \leftarrow P$ $\mathcal{E}.G_S \leftarrow p.G_S$

Here we use $\mathcal{P}$ as the set of allowed process names from which the existing set is subtracted and a random choice is made and assigned to P. This ensures unique process variable names. The variable P is added to the set of unguarded processes. Finally, after p is generated, we take the resulting set of guarded processes and return it.

It is important to note the dependencies of the attributes. The occurrence of a process variable depends on the value for G_S for its binding equation. This means the generation of occurrences of process variable names is postponed until everything but the occurrences are generated in the process expressions.

7 Directed Testing

In general and specifically in the mCRL2 toolset, tools consist of the implementation of one or more algorithms that operate on certain input. Commonly, these algorithms operate of specific parts of this input. For example, tools for PBESs contain algorithms that operate on parts of this PBES like the fix-point equations, quantification operators, parameter names, etc. So these algorithms are designed to work on parts of the input string, possibly on specific grammatical structures.

If these algorithms are to be tested, it can be useful to put emphasis on the particular grammatical structures. If an algorithm is specified to reduce the number of parameters, a parameterless input string will not serve as useful test input for this algorithm. If this algorithm contains an error, i.e. incorrect reduction of the number of parameters, a parameterless input string will not reproduce this error. Furthermore, it is possible that the probability of the reproduction of the error is related to the amount of parameters in the input string.

In directed testing, we guide the generation of strings such that the resulting strings have a higher probability of having certain language patterns, e.g. by taking away some of the non-determinism in production rule choices. For this we have introduced weights, but we can also do this using constraints: we can specify attribute rules and guards that ensure the choice of certain production rules. We can also use constraints to force a specific amount of a structure, described in the following section, and even directly produce certain language patterns.

In section 4.1.2 we have used weights to influence the generation of certain grammatical structures. However, weights have in fact little influence on the number of applications of a repetition. To accommodate for this, we can design a constraint where this structure frequency can be manipulated.

Consider the following production rules that produce a sequence of one or more as. The constraint we will define resembles example 5.2.1: we count the amount of repetitions we still need to reach a predefined amount.

$$\begin{aligned} S &\to T \\ T_0 &\to aT_1 \mid a \end{aligned} \tag{7.1}$$

Non-terminal T will be assigned an inherited attribute $Number$. We set the initial value for $Number$ through the first production rule. Guards will force the choice of a single a if the value is 1. If the value is greater than 1 the guards will force a recursion through T.

Production rule	*Attribute rules*	*Guard*
$S \to T$	$T.Number \leftarrow n$	
$T \to a$		$T.Number = 1$
$T_0 \to aT_1$	$T_1.Number \leftarrow T_0.Number - 1$	$T.Number > 1$

Here, $n \geq 1$ is the predefined amount of occurrences for a. This shows how structure frequency can be modeled using constraints.

8 Test Case Design

8.1 Partial Testing

In black-box testing where we have output from a program we can only verify it through a specification of the program's functionality and the corresponding input. The verifier of the result of the execution of a test is commonly called a *test oracle.* A re-implementation of the specification can serve as an oracle. However this is not feasible since specifications are not always present. Furthermore, specifications can be complex and most tools have undergone years of development making reimplementations error prone.

Instead, we opt to use other tools from the toolset to test for a property or invariant of a tool, i.e. partial testing. For tools where For example, consider a tool is specified to transform an input while preserving some property. If there is a second tool that can decide whether or not two objects are equal in that property, it can be used on the input and output of the first tool to check the correctness of the first tool. Now the tests succeed, as long as the second tool produces the same answer for every combination of input and corresponding output of the first tool.

As stated before, tools in the mCRL2 toolset require one or more input strings that are specified through a context-free grammar. The same holds for the output of the tools, although there are some that only produce side effects, like a visualization program. The programs also require values for parameters that influence the functionality. The definition below formalizes a tool, so we can use them to define tests.

Definition 8.1.1 (**Tool**)**.** A tool is a four-tuple $T = (I, O, A, P)$ where

- I is the list of input grammars of the tool
- O is the list of output grammars of the tool
- A is the set of parameters and their values
- P is the tool's functionality relating input and output

The choice for lists is made because there is a distinct ordering on input and output. Consider a tool $T = (I, O, A, P)$ where the input consists of two terms from different languages G_A and G_B. Then $I = [\ G_A, G_B\ ]$, so a list of the two grammars, if the first input corresponds to grammar G_A and the second to G_B. So the definition of tools ensures an ordering in input and output and this also applies whenever there are multiple inputs that specify the same grammar. We abstract from the functionality P, since we are not interested in white box testing.

In the previous example of a tool deciding the output of another tool, we saw in fact the application of a tool on the output of another. We can view a test case as a set of tools, some of which use the output of others as input, along with an expression over one (or more) of the outputs to determine the test result, i.e. the testing oracle. This reduces the risk of errors in the verification while providing an efficient way to specify test cases. When an error is found, if the oracle is correct, one of the programs in the set of tools contains an error.

The set of tools in a test case can be used as nodes in a graph and by doing this a test case can be considered as a graph consisting of tools linked together through intermediate 'grammar' nodes.

Figure 8.1 displays a tool connected to its input and output nodes. For grammar nodes $l_1, ..., l_6$, the grammar type of l_i is G_i for some context-free grammar G_i. Different grammar nodes can have the same grammar. The nodes themselves have a label so a distinction can be made between two grammar nodes that have the same grammar.

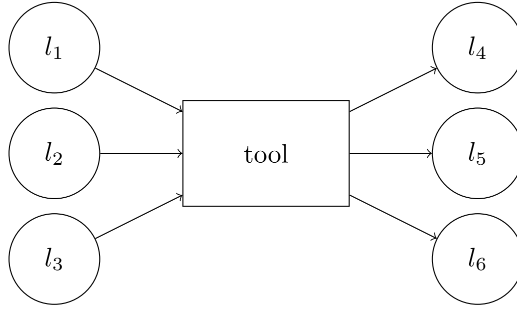


Figure 8.1: A tool with three input and three output nodes that have a language type

During the execution of the test, each grammar node will be given a value. This value can be reused by more tools, but not altered. This means every grammar node has at most one incoming edge, from the tool that will produce its value as output. Additionally every instance of a tool can only be applied once during the test. We should also enforce every input and output of every tool in the graph to be mapped to a grammar node with the correct grammar label.

Definition 8.1.2 (**Test case**). A *test* is a four-tuple (V_T, V_L, E, F) where

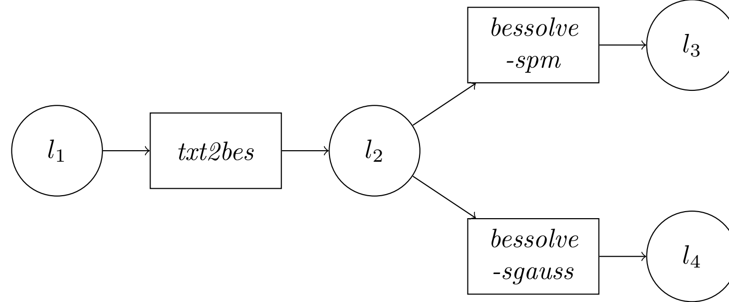
1. $G = (V, E)$ with $V = V_L \cup V_T$ is a directed acyclic connected graph
2. $V_T = \{t_1, ..., t_n\}$ is a set of tools, so $t_i = (I_i, O_i, A_i, P_i)$, for $1 \leq i \leq n$
3. $V_L = \{l_1, ..., l_m\}$ is a set of nodes with a grammar type:
   - $\gamma(l_i) = G_i$ for some context-free grammar G_i and $1 \leq i \leq m$
4. there are only edges between different node types, $E \subseteq (V_L \times V_T) \cup (V_T \times V_L)$
5. every node $v \in V_L$ has at most one incoming edge,
   - $(\forall l, l' \in V_L, t, t' \in V_T : (t, l), (t', l') \in E : l = l' \Rightarrow t = t')$
6. every input and output of every tool is mapped to a node, so
   - $(\forall t_i \in V_T : (\forall j \in I_i : (\exists (l, t_i) \in E : \lambda(l) = j)))$
   - $(\forall t_i \in V_T : (\forall j \in O_i : (\exists (t_i, l) \in E : \lambda(l) = j)))$
7. a verdict $F : L(G_1) \times ... \times L(G_m) \rightarrow \mathbb{B}$ is a Boolean function over strings from the languages defined by the types of the grammar nodes

There are some grammar nodes that will serve as the starting point of our tests, i.e. when these nodes are given a value one of the tools can be applied and the testing can begin. These nodes will comprise the initial configuration, which will be the set of nodes that do not have an incoming edge. Finally there has to be a set of end nodes for which it can be decided whether or not the test succeeded or failed, based on the values they will get during execution of the test.

Example 8.1.1 (**BES Solver**). *A solver for BESs is implemented in the tool* bessolve. *This solver outputs either true or false, depending on the solution, if there indeed is a solution for the input BES. Two solving strategies are available in* bessolve, *namely small progress measures (SPM) and Gaussian elimination. Both of these strategies are specified to give the same answer for every BES. We can use this invariant and design the following test case to test if the tools*

respect the invariant.

The tool txt2bes *is there to transform the textual representation of a BES into a mCRL2 internal representation. The internal representation is the required representation of the input for most tools in the mCRL2 toolset.*



If we generate a random BES for l_1 and run the test by executing the tools. The values for l_3 and l_4 should always be equal. If we have a test run where this is not the case, we have encountered an error in one of the tools of this test case along with the input serve as a counterexample.

The visualization of a test case as a graph in example 8.1.1 is used to define these test cases in this document. The types of the grammar nodes are omitted, since they can be inferred from the tools. The verification expression is informally described in the accompanying text.

8.2 Execution

The execution of a test case involves running the tools as specified by the test case while directing the input and output between them. By giving values to language nodes we enable the execution of a tool. We can define the Boolean function *enabled*(t_i) to be true when all input nodes for t_i have a value. The necessary actions for a test case execution is described in the following algorithm.

ExecuteTestCase(V_T, V_L, E, F)
1 **for** $l_i \in V_L$ that has no incoming edge
2 **do** generate random string value for l_i based on its grammar
3 **while** $V_T \neq \emptyset$
4 **do** take tool $t_i \in V_T$ for which *enabled*(t_i) is true
5 execute t_i and put the output in the corresponding node(s)
6 remove t_i from V_T
7 **return** F applied to the values for nodes $l_i \in V_L$

The successful run of this algorithm and thus the execution of a test case depends on the correctness of the test case with respect to definition 8.1.2. So every input and output of every tool is mapped to a node and for every tool, its input nodes eventually get a value so every tool is eventually enabled. When a test is executed, not all tools will successfully terminate. Even if it is presented with valid input, the tool could error or timeout, which makes it unable to verify the end result of a test. For every test run, the following results are possible.

- **Tool timeout**
  Depending on the input a tool could fail to terminate within a specified amount of time, which will cause a time out to occur. Commonly this is not a result of an error in the tool

but it is indeed expected behaviour. For this reason, whenever a timeout is occurred, the current test run is discarded. We can then continue to generate new input and do another test run. Of course, it can be the case that a timeout is not expected behaviour in which case we can count the test result as a failure and conclude we have discovered an error.

- **Tool error**
  A tool can return an error message whenever it is presented with invalid input. Since we want to test tools by giving them valid input, this error message is an indication of a missing constraint.

- **Result success**
  If the test result is verified and considered valid, we can conclude the tools preserve the invariant under test for a single set of generated input. We can subsequently discard the result.

- **Result failure**
  If the test verdict produces false, we have an instance where the invariant under test is violated. This means that not only is there an error in one of the tools of the test case, but we also have a counterexample (the generated input), which can be used to reproduce and localize the error.

The result failures are the most interesting result. If a result is false, it means that the tools have ran successfully but the invariant under test has been violated. So this indicates that we have found an error and we have the corresponding input to serve as a counterexample.

Tool errors are indication of a missing constraint. Depending on the frequency of a generated input being invalid with respect to a semantic requirement, it is necessary to add a constraint to solve this. If the probability that a generated string violates a semantic requirement is significantly large, a constraint has to be added.

9 Implementation

Until now we have seen the complete approach to the random testing of the mCRL2 toolset. The concepts of string generation with constraints, as well as the approach to building test cases are combined and implemented into a testing framework. This framework is written in Python, which is the standard scripting language used in the mCRL2 toolset.

In the framework the algorithms GenerateII and ExecuteTestCase are implemented. A parser for the mCRL2 grammars is used to import the context-free grammars into a datastructure of multidimensional lists. In the framework the functions f and f' are also implemented, which operate on this datastructure.

The framework operates on the context-free grammars in the mCRL2 toolset in BNF. The production rules in these grammars are labelled. Separate from the grammar a set of constraints is built for every grammar. These constraints are annotations to the production rules in the context-free grammars, using the labels to refer to the production rules. The following annotations are supported in the constraints.

- **Weight**
  The weight of a production rule, described in section 4.1.2, affects the probability of that production rule being chosen by the algorithm to apply to a symbol.
- **Attributes**
  The set of inherited and synthesized attributes can be defined per production rule. Since these sets pertain to a non-terminal and not a production rule, these should be the same for every production rule with the same non-terminal as its left-hand side.
- **Attribute rules**
  The set of attribute rules as defined in definition 5.2.1, that determine the value of the attributes for every symbol in the rule. They define the values for the inherited attributes of the right-hand side non-terminals and the values for the synthesized attributes of the left-hand side non-terminal. For tokens there is an attribute rule that defines the string value for that token.
- **Guard**
  A single Boolean expression serves as guard of a production rule (see definition 5.2.1). This guard can also be omitted, in which case the algorithm will default the evaluation to true.
- **Attribute dependency**
  A production rule can be marked if it has an attribute dependency that cannot be respected in a left-to-right traversal strategy. When a production rule is marked, the application is postponed until everything else is processed, after which the marked symbols are processed.

The implementation of GenerateII means we have a left-right processing of symbols in production rules. Some static semantics, e.g. variable binding, have different dependencies. Rather than rewriting the grammar (which is not always possible), we opt to postpone the processing of a symbol until the rest is processed.

So after the first pass of the algorithm we have a term that contains several marked symbols and for the rest solely terminal symbols. The context returned by the first pass is then used in the second pass to process the marked symbols.

These annotations are grouped by production rule (using the labels from the grammars). Every grammar has several semantic requirements and the constraints that fulfill these requirements are combined as annotations in a single constraints file.

The grammar and constraints are separated which allows the user to create different sets of constraints on the same context-free grammar, without having to touch this grammar. This is useful in cases where the user wants to switch between completely different sets of constraints. However, since the constraints are combinations of several semantic requirements, the user has to copy the relevant parts when he wants to reuse them.

A database contains the test cases designed per definition 8.1.2. Every test case contains the graph in a textual format, the grammars and constraints needed for input generation, as well as a small block of code that serves as an oracle. This code is executed to determine the result of the test case, where it is allowed to use the values for the nodes in the test case. These values are calculated during test execution.

The user of the framework can specify the set of tools for which the test cases should be executed. This allows the user to test several versions of the same tools without having to modify the test case. Files serve as input and output for the tools in a test run. Every string from the language of one of the formalisms of mCRL2 is recorded in a separate file. When a test run is unsuccessful, the corresponding input that is stored in a file is kept. For successful runs, all the input and output is discarded, unless specified otherwise by the user.

The user can run a test by importing the test case, the grammar and constraints, by specifying the location of the tools and by specifying the given input string generation length, number of tests to run and the timeout time. The framework will calculate the necessary table entries for f and f' after which it runs $ExecuteTestCase$ the specified number of times. Exceptions are raised when a tool errors or times out, which the user can choose to handle or ignore.

Every constraint from section 6 is implemented. Appendix A shows the context-free grammars for which constraints are implemented. For BESs this is the entire grammar. For PBESs and mCRL2 process specifications, a subset of the data grammar is implemented. This subset pertains to the basic types of the data grammar. Furthermore, the grammar for mCRL2 process specifications is not complete as it misses operators, like renaming and hiding actions. The main reason for their omission is the amount of work: every operator and data type can have a multitude of constraints which have to be manually constructed.

For the remainder of the grammar and constraints, implementation should be straightforward. However, for the support of composite types like lists and sets or user-defined types, the domain of the typing attribute can become complex. The reason is that these compositions produce an infinite set of possible types and the attribute rules and guards have to support this to generate type correct strings.

The framework logs the result of every test execution. The corresponding generated input is only recorded when the result is false.

10 Experimental Setup

This section describes the setup of the experiments done in an attempt to achieve the main goal presented in this thesis, which is finding new errors in the mCRL2 toolset. Before we apply the presented method to the latest version of the toolset, we can first determine if the presented approach is effective at finding errors at all. Furthermore, we want to find an effective testing strategy that we can apply. There is one parameter, namely the length for string generation, that we can vary. Therefore it is useful to determine actual length of the resulting strings.

All in all we can identify the following testing goals.

- Validate the approach presented in this thesis by applying it to previous cases
- Determine the actual length of generated strings
- Find a strategy for random testing by determining the optimal testing string length per formalism and duration of tests
- Apply this strategy on the latest release in an attempt to discover new errors
- Apply directed testing to increase the error probability

Currently there is an existing random testing method available in the mCRL2 toolset. This testing method uses a bottom-up approach to string generation which differs only in implementation details from our top-down approach. The existing method is ad hoc; for every grammar there exists a separate module that can build a string from (a subset of) the language defined by this grammar, in contrast to our approach where the input is a grammar. There is however a mechanism in the existing method that aims to minimize the error-producing input strings. Having the minimal length input that produces an error can be very useful for fixing the error.

The existing method has a set of parameters that determine the amount of several grammatical structures, e.g. the number of fix-point equations, in the resulting string. In our approach this is done using constraints. The existing method does not enable the weighting of production rules.

The existing random testing method has identified several errors in previous versions of the toolset. We can apply the presented method implemented in our framework to these previous cases in attempt to verify the effectiveness. Our framework should be able to find any error previously discovered by the existing random testing method.

10.1 Validation

In an attempt to verify the random testing approach presented in this thesis, we apply the method to previous versions of the mCRL2 toolset. In several revisions, there have been fixes for errors discovered by the existing random testing method. If we apply our method to the revisions before the ones that introduce these fixes, we should be able to encounter those errors again. We can subsequently verify the effectiveness of our method.

So we aim to reproduce the tests used to find the errors in previous revisions of the mCRL2 toolset. We can use the following test plan.

- We search for revisions in which a fix is introduced for an error that was identified through random testing. This fix can span a number of subsequent revisions, as it can take multiple revisions to resolve an error. Since it is unknown which revision actually introduces the error, we take the revision before the first that is part of the series with the fix. This gives us a number of revisions and errors to test for.

- For the tool that contains the error in a selected revision, we design a test case per definition 8.1.2 to test for an invariant of that tool. This is the same test case as the one that discovered the error in the original random testing method. This gives us a test case for each revision.
- We identify the grammar and set of constraints that have to be imposed on that grammar to produce valid input for our test cases. The grammars for the formalisms can be different from one revision to another, so when testing a specific revision, we use the grammars from that revision.
- We test for one hour per length, varying the length between 30, 60, 120 and 240. Since the running time increases linearly with the length, we increase the length exponentially.
- For each test run of an hour we record the amount of single test cases ran, the error frequency, number of timeouts and every error-producing input string. We can use all of this information to determine an optimal testing strategy.

Several revisions will not be suitable for selection. These revisions are regarded unstable and include revisions where some tools are not able to compile. Also revisions where regression tests fail can be unsuitable. The reason is that failing regressions tests can indicate unexpected behaviour in tools, so depending on for which tools the regression tests fail we can choose to exclude the revision.

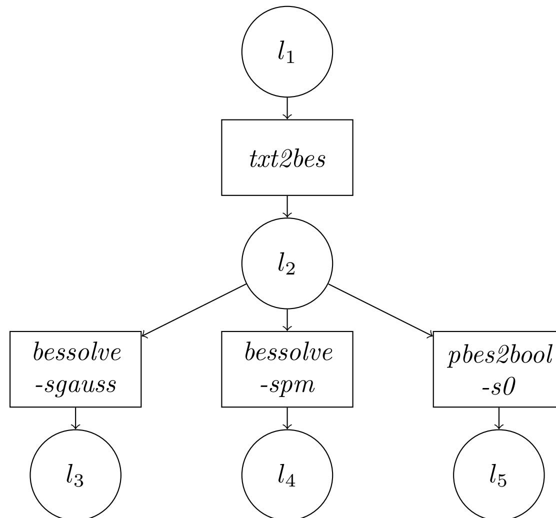
It is possible for a tool to time out. For example a PBES solver that cannot come to a solution due to infinite data domains will run indefinitely. We set a limit to the execution time of a tool, so a timeout occurs whenever a tool exceeds this time limit. We can empirically determine this limit by sampling a number of test runs and examine the execution time. Depending on the distribution of the running times we can take a limit thats above a majority of run times. In practice, when the tools are supposed to terminate on their input, the running times will be close together so choosing a limit is straightforward.

When a timeout occurs during execution of one of the tools of a test case, this test run is not counted towards the total number of executed tests. Note that this does not mean the test result is a failure. Instead, whenever a timeout occurs, the test is reran. The test and timeout time do count towards the hour testing time.

To acknowledge that the test failures are indeed due to the error under test and that the test results are not diluted by other errors, we manually verify every failed test. When we encounter a large number of failures (e.g. more than 100), we take a subsample of 10% to manually verify.

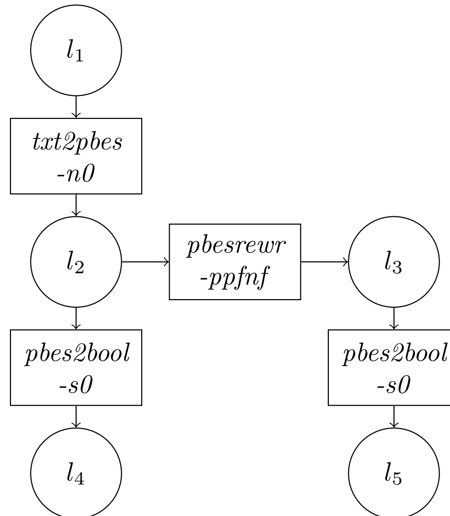
Listed below are the revisions used for testing. For every revision we have a test case in the form of a graph and a description of the tool and invariant under test as well as a specification of the end result.

- **Revision 8316 (bessolve)**
  In this revision there is an error in the small progress measures algorithm that is implemented in the tool *bessolve*. This tool takes a BES as input and solves it, outputting either true or false. We design a test case where a random BES is solved in three ways: using the tool *bessolve* with Gaussian elimination (*-sgauss*) and small progress measures (*-spm*) and using the tool *pbes2bool* with the default strategy (*-s0*). The following graph displays this test case.



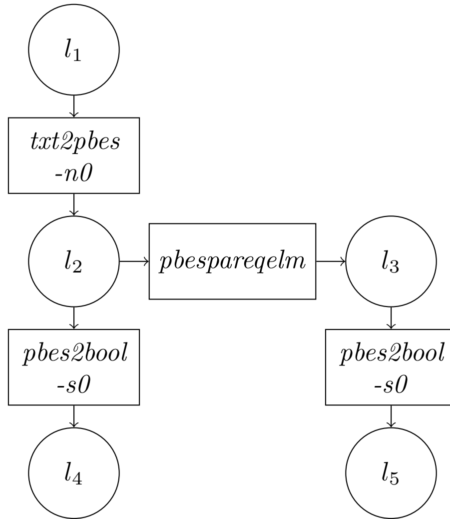
Every solver should give the same answer for the same BES. This means the test will succeed when the Boolean values for l_3, l_4 and l_5 are equal.

- **Revision 8693 (pbesrewr)**
  The test case for revision 8693 involves a PBES that is rewritten into PFNF, a normal form, through the tool *pbesrewr* with (*-ppfnf*). The original and resulting PBES are solved through the tool *pbes2bool*. The solutions should always match. The following graph displays this test case.



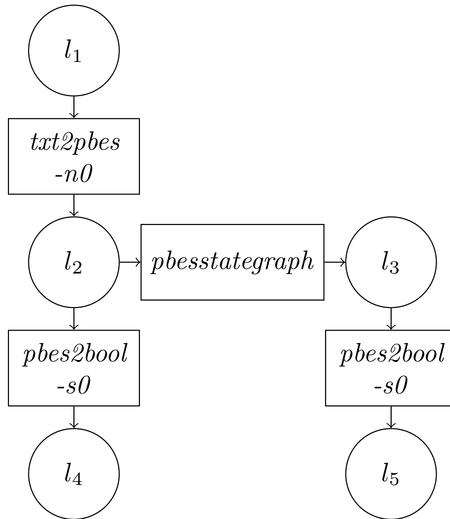
  The test will succeed if and only if the Boolean values for l_4 and l_5 are equal.

- **Revision 9390 (pbespareqelm)**
  The test case for revision 9390 involves a PBES that is transformed using *pbespareqelm*, while the solution is specified to be preserved. The original and resulting PBES are solved through the tool *pbes2bool*. The solutions should always match. The following graph displays this test case.



The test will succeed if and only if the Boolean values for l_4 and l_5 are equal.

- **Revision 11070 (pbesstategraph)**
  The test case for revision 11070 involves a PBES that is transformed using *pbesstategraph*, while the solution is specified to be preserved. The original and resulting PBES are solved through the tool *pbes2bool*. The solutions should always match. The following graph displays this test case.



The test will succeed if and only if the Boolean values for l_4 and l_5 are equal.

11 Results

From section 5.3 we know that the resulting length can differ from the given length. To get an indication of what this difference is, we experiment by generating PBESs with three lengths 60, 120 and 240 and record the resulting lengths. Figure 11.1 shows the results of this experiment.

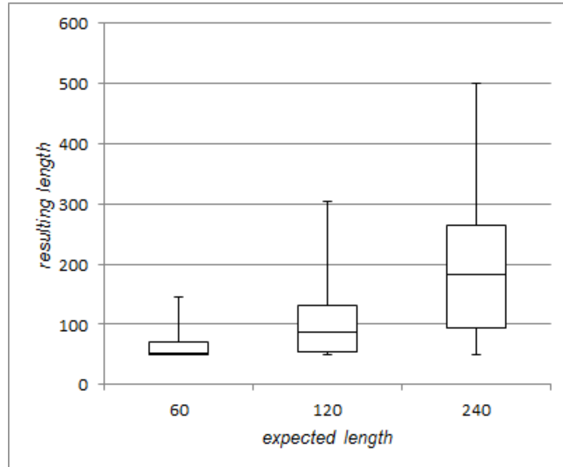


Figure 11.1: Expected lengths versus resulting lengths for PBES generation with constraints.

From the results we can see that the resulting length can differ significantly from the given length. The majority of resulting lengths are smaller than the given length, but can be worst-case doubled or worse. This large difference makes length not a useful metric for generated strings, outside of its purpose for termination in the string generation algorithm.

11.1 Validation

This section describes the results obtained from the application of the presented random testing method, implemented in a framework, to the previous revisions of the mCRL2 toolset. The tables listed below show the data obtained from the test runs. Per revision we can see the amount of tests ran in an hour, the number of error occurrences and the amount of timeouts that have occurred. If the error occurrence is zero, it means that the framework was unsuccessful in finding the error. From the results we can see that the framework can find every error for all test cases of previously discovered errors. This verifies the effectiveness of the presented approach.

- **Revision 8631**

Length	Tests	Errors	Timeouts
30	38226	36181	11
60	28000	27746	10
120	18365	18342	19
240	11011	11005	6

In this revision we have tested errors in the BES solver algorithms. We can see a very large fraction of test cases where an error has occurred. This can be explained due to the fact that the small progress measures algorithm in the *bessolve* tool was still under development. The ratio of failed tests increases as the strings become longer.

- **Revision 8693**

Length	Tests	Errors	Timeouts
30	35510	0	1
60	25074	5	21
120	13938	5	28
240	8903	8	45

  This revision contains an error in the PFNF rewriter. The results show that for a length of 30 we have been unsuccessful at finding the error. For the other lengths however, the error has been found. The highest error ratio is with the highest length. This can be explained due to the fact that when a string is longer, it has higher probability of containing the error producing structure.

- **Revision 9390**

Length	Tests	Errors	Timeouts
30	27624	0	0
60	22514	1	3
120	13597	3	17
240	8769	4	16

  In this revision we have tested the *pbespareqelm* tool. From the results we can conclude we have been successful at identifying the error. Again, for length 30 however, there have been no test failures. Apparently this length is too small for the error producing grammatical structures to occur, as we can see the frequency increasing with the length. The error ratio is low, which can be expected when the error is relatively hard to find.

- **Revision 11070**

Length	Tests	Errors	Timeouts
30	20677	32	1
60	16491	67	8
120	11065	55	14
240	6032	45	16

  This revision pertains to the *pbesstategraph* error. The results show that we can find the error at every tested length. We can see the amount of tests decreases exponentially with the length increasing while the amount of timeouts grows slightly. The most interesting about these results is the fact that a length of 60 actually has the highest error frequency. However, the error ratio is the highest with a length of 240. This means the occurrence of the error is due to a particular grammatical structure in the resulting string. The larger the string, the higher the probability it contains this grammatical structure.

The results for several revisions show an increase in the number of errors when the length increases. In the tests for revision 9390 we can see the errors increase with the length. It is interesting to see if this holds for even greater lengths. Since an increase in length results in fewer tests per hour, the number of errors has to decrease at some point.

We repeat the experiments for revision 9390 with two additional lengths. Table 11.1 shows the results for lengths 480, 960. The results show already for length 480 that the number of errors is not increasing.

Table 11.1: Test results for an hour test run per length for revision 9390.

Length	Tests	Errors	Timeouts
30	30932	0	4
60	23068	3	4
120	12827	3	12
240	8944	4	24
480	4876	3	16
960	1869	2	48

11.2 New Errors

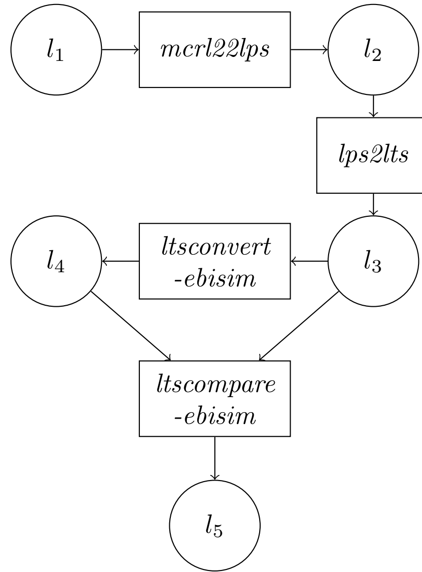
Using the results obtained from the validation experiments, we can determine a testing strategy in order to find new errors. For sufficiently large length, we have discovered every error within the one hour testing time. Therefore we take the same testing time in our strategy. From the results we cannot conclude anything about the relation between length and the amount of errors, except that a length of 30 appears to be too small. Therefore we take a length of 120 for every test case.

The test cases we use are listed below. We take the four test cases from the validation experiments (bessolve, pbesrewr, pbespareqelm and pbesstategraph) and add a number of test cases for linear process specifications (LPSs). The majority of test cases involves a transformation that preserves a solution (in the case of BES and PBES) or bisimulation (in the case of LTSs and LPSs). The original and transformed term are compared in terms of their solution or their bisimulation equivalence. The following list displays the test cases for which we run the experiments.

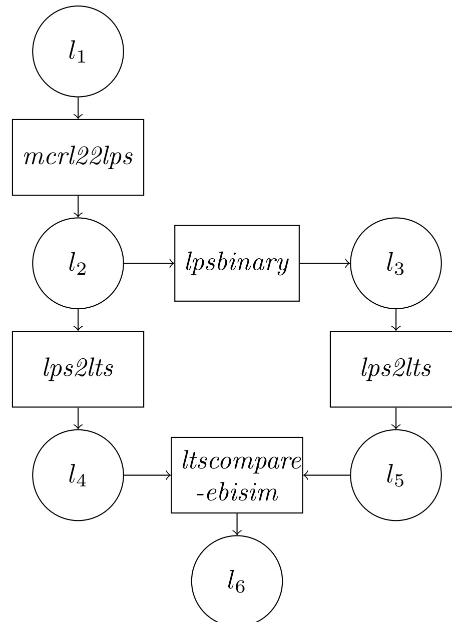
Test case	*Description*
bessolve	comparison of BES solving algorithms
pbesrewr	PBES transformation and two-way solving
pbespareqelm	PBES transformation and two-way solving
pbesstategraph	PBES transformation and two-way solving
lpsbinary	LPS transformation and two-way bisimulation test
lpsparelm	LPS transformation and two-way bisimulation test
lpsconstelm	LPS transformation and two-way bisimulation test
lpssumelm	LPS transformation and two-way bisimulation test
lpssuminst	LPS transformation and two-way bisimulation test
ltsconvert	LTS transformation and two-way bisimulation test

The tools that work with LPSs will be supplied a random LPS by generating a random mCRL2 process specification and using the tool *mcrl22lps*. The following graphs show this while displaying the test cases for *lpsbinary* and *ltsconvert*.

- **ltsconvert** This test case involves a random mCRL2 process specification converted into an labelled transition system (LTS). This LTS is then converted while preserving the bisimulation equivalence relation. The *ltscompare* tool decides whether or not the two input LTSs are bisimilar. The invariant under test is that *ltsconvert -ebisim* outputs an LTS that is bisimilar to the input LTS. This means the value for l_5 should always be true.



- **lpsbinary**
  In one of the more recent revisions an error has been identified which could be a candidate for random testing. The error involves the tool *lpsbinary* which does not preserve strong bisimulation when presented with at least one identified counterexample. Since this is an invariant for *lpsbinary*, it is interesting to see if we can find this error through random testing. The following graph displays the complete test case.



A mCRL2 process specification is transformed into a linear process specification (LPS) after which *lpsbinary* creates another LPS from it. Both are converted into a labelled transition system (LPS) and compared together using *ltscompare*. With *-ebisim* a comparison is made based on the bisimulation equation relation. For every mCRL2 process specification for l_1, l_6 should get the end result true.

The test cases are applied to the several release versions of the mCRL2 toolset, namely 201310.0 (October 2013), 201210.0 (October 2012) and 201202.0 (February 2012). For every hour of test runs we record the amount of tests, the amount of errors found and the amount of timeouts that have occurred. Tables 11.2, 11.3 and 11.4 displays the results of the test runs for the release versions.

Table 11.2: Test results for an hour test run per test case on mCRL2 version 201310.0

Test case	*Tests*	*Errors*	*Timeouts*
bessolve	22636	0	15
pbesrewr	17048	0	19
pbespareqelm	15076	0	20
pbesstategraph	10123	0	9
lpsbinary	9479	0	476
lpsparelm	9969	0	458
lpsconstelm	8431	0	369
lpssumelm	11157	0	415
lpssuminst	8051	0	531
ltsconvert	8905	0	514

Table 11.3: Test results for an hour test run per test case on mCRL2 version 201210.0

Test case	*Tests*	*Errors*	*Timeouts*
pbesrewr	16791	0	9
lpsbinary	7380	0	326
lpsparelm	7529	0	351
lpsconstelm	7186	0	355
lpssumelm	7209	0	369
lpssuminst	7677	0	376
ltsconvert	7004	0	363
lpspp	6881	0	317

Table 11.4: Test results for an hour test run per test case on mCRL2 version 201202.0

Test case	*Tests*	*Errors*	*Timeouts*
lpsbinary	7942	0	351
lpsparelm	7360	0	333
lpsconstelm	5608	0	280
lpssumelm	6090	0	265
lpssuminst	6119	0	258
ltsconvert	8301	0	341
lpspp	6729	0	339

The results show that we have been unable to discover any new errors for these test cases. This could be due to the fact that there are no errors left. In fact, several of the tools in these test cases have been tested before using the existing random testing method. We can identify four main reasons for the fact that we have not found a new error.

- Since we use a subset of the data grammar, it could be the case that in the remaining grammar there is an error.
- The probability of any remaining errors occurring is such that an hour of testing is not enough to find them.
- For the invariants under test, the tools are error-free.

It is important to note the relatively large amount of timeouts that occur for test cases working with mCRL2 process specifications.

11.3 Directed Testing

To experiment with directed testing (see section 7) we take a test case for which we know what the error producing language pattern is. In the *pbesrewr* tool in revision 8693, there is an issue with name clashes of variables in quantification operators. Consider for example the following PBES.

$$(\nu X = (\forall u \in \mathbb{N}.u < 5) \wedge (\exists u \in \mathbb{P}.u > 0)) \tag{11.1}$$

This single fix-point equation contains two quantifications that both use the variable u. Since they are in different scopes, this should be no issue. However, the tool *pbesrewr* in revision 8693, contains does not handle name clashes like this correctly.

We want to put an emphasis on generating these quantifications. Therefore we opt to use a constraint that sets the amount of generated fix-point equations to 1 and set the weight of the production rules producing quantifications to 10.

Scenario	Length	Tests	Errors	Timeouts
1	120	13938	5	28
2	120	6450	23	15

In the above table we compare the results of the experiments without directed testing (no. 1), with the scenario for directed testing described before (no. 2), for length 120. From the results we can see that the amount of errors that have been found has increased from 5 to 23. We can also see the number of tests has decreased, which can be explained due to the constraint for a single fix-point equation. This constraint will force the algorithm to choose new lengths regularly, which increases the generation time.

12 Conclusion

In this work an approach for the random testing of the mCRL2 toolset using attribute grammars is presented. Starting from the basic context-free grammars defining the language of the mCRL2 formalisms, we used a length-controlled string generation algorithm to generate input for the tools of the toolset. To prevent the generation of meaningless input, we enhanced the context-free grammars with attributes, rules and guards, turning them into attribute grammars. For several semantic requirements we have seen attribute grammars that force strings from the language defined by that grammar, to adhere to these semantic requirements.

We have used an abstract and formal definition of attribute grammars and related concepts and have used a generic approach to building constraints. This generic approach enables the presented methods to be applied in any other situation of context-free grammars and semantic requirements. We have presented an abstract definition for building test cases where invariants of tools are tested. These invariants come from the specification of the tools and enable us to do partial testing of the tools.

The length of a string is used successfully as a control mechanism. However, outside of termination this length is not a useful metric. The reason is that the resulting length can differ significantly from the given length and the way it is used forces a certain traversal strategy (left-right).

All these concepts are implemented into a testing framework. To validate the effectiveness of the presented approach we have used the framework on previous versions of the mCRL2 toolset. These versions contain errors which were once discovered by an existing random testing framework. From the results we can conclude that the presented approach is effective in finding errors: for every test case the framework was able to find the error.

A small experiment for directed testing has shown how a probability of an error occurring can sometimes be increased. When an error-triggering pattern in a string is known, we can use constraints to generate this pattern with a higher probability. Subsequently, the error should occur with a higher probability.

Using the results obtained in the validation experiments, we have determined a testing strategy which we employed in order to find new errors. For the test cases that we have used, we did not discover any errors. From this we cannot conclude that there are no errors left, since it could be the case that there are errors left that have such a low probability of being occurring and the testing amount has not been sufficient. Furthermore, only a subset of the data and mCRL2 process specification grammars are implemented. It could be the case that by implementing the remaining grammar and constraints errors can be found.

To increase the effectiveness of the testing framework this is also the main consideration: implementing the remaining grammar and constraints will have the largest increase of the effectiveness of the testing framework. Additionally, more tests could be designed especially for the less commonly used tools.

13 Future Work

13.1 Method

The length-controlled string generation algorithms we presented employ a left-to-right traversal strategy, postponing the processing of a symbol when its attribute dependencies do not occur to its left. This has been sufficient for the grammars and constraints in the test cases we considered. However, it is possible that other grammars or semantic requirements cannot be handled by this traversal strategy. So future work includes incorporating more traversal strategies and possibly investigating the effect of different traversal strategies on string generation.

We have used the length metric as a control mechanism. However, there are several other control mechanisms that can be considered ([21]). Even though these control mechanisms will be similar in their usage and limitations, it can be useful to investigate their differences for string generation. Furthermore, future work could include research on how control mechanisms can be used with directed testing.

13.2 Experimentation

The list of test cases we have used in the experiments are non-exhaustive. There is a multitude of other tools and invariants for which test cases can be built. Since these test cases could reveal errors that were previously undiscovered, it is useful to use the framework with these new test cases in the future.

For the formalisms using data, we have used a subset of the grammar pertaining to the data. This includes PBESs and mCRL2 process specifications. For future experimentation with the random testing framework, it can be useful to extend this data grammar to allow for, for example, user defined types.

Several other grammars from mCRL2 are not supported in the random testing framework. These are linear process specifications (LPS), labelled transition systems (LTS) and mu-calculus formulas. These formalisms have their own tools, which leads to new test cases. Since a randomly generated mCRL2 specification can be transformed into an LPS and LTS we can in fact test with these formalisms. Still, these grammars could be included into the framework in the future.

Though we have experimented with directed testing, we have not employed it yet in our testing strategy. Depending on the test cases, it can be useful to look for new errors using directed testing, e.g. by varying the frequency of certain grammatical structures.

Bibliography

[1] Dániel Drienyovszky, Dániel Horpácsi, and Simon Thompson. Quickchecking refactoring tools. In *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang*, Erlang '10, pages 75–80, New York, NY, USA, 2010. ACM.

[2] A. G. Duncan and J. S. Hutchison. Using attributed grammars to test designs and implementations. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 170–178, Piscataway, NJ, USA, 1981. IEEE Press.

[3] Jan Friso Groote and Mohammad Reza Mousavi. *Modelling and Analysis of Communicating Systems.* Department of Computer Science, Eindhoven University of Technology, 2010.

[4] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[5] Darrel C. Ince. The automatic generation of test data. *The Computer Journal*, 30(1):63–69, 1987.

[6] U. Kastens. *Ordered attributed grammars.* Fakultät für Informatik: Interner Bericht. 1978.

[7] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Correction: sl Mathematical Systems Theory 5, 1, pp. 95-96 (March 1971).

[8] Ralf Lämmel and Wolfram Schulte. Controllable combinatorial coverage in grammar-based testing. In *In TestCom*, 2006.

[9] Peter M. Maurer. The design and implementation of a grammar-based data generator. *Software Practice and Experience*, 22:223–244, 1992.

[10] Bruce McKenzie. Generating strings at random from a context free grammar, 1997.

[11] Norman Paterson. *Genetic programming with context-sensitive grammars.* PhD thesis, Saint Andrew's University, September 2002.

[12] A. S. M. Sajeev and Sita Ramakrishnan. Use of attribute grammars in test-sequence specifications, 1997.

[13] M.L. Scott. *Programming Language Pragmatics.* Elsevier Science, 2009.

[14] Arindama Singh. *Elements of Computation Theory.* Texts in Computer Science. Springer, 2009.

[15] Michael Sipser. *Introduction to the Theory of Computation*, chapter Context-free Languages. International Thomson Publishing, 1st edition, 1996.

[16] Emin Gün Sirer and Brian N. Bershad. Using production grammars in software testing. *SIGPLAN Not.*, 35(1):1–13, December 1999.

[17] K. Slonneger and B.L. Kurtz. *Formal syntax and semantics of programming languages: a laboratory based approach.* Addison-Wesley Pub. Co., 1995.

[18] Bart Vergauwen and Johan Lewi. A linear algorithm for solving fixed-point equations on transition systems. In *Proceedings of the 17th Colloquium on Trees in Algebra and Programming*, CAAP '92, pages 322–341, London, UK, UK, 1992. Springer-Verlag.

[19] William M. Waite and Gerhard Goos. *Compiler Construction.* 1996.

[20] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20(11):822–823, November 1977.

[21] Vadim Zaytsev. Combinatorial Test Set Generation: Concepts, Implementation, Case Study, June 2004.

A Context-Free Grammars

This section lists the context-free grammars that are used to generate input for the tools that operate on these formalisms. The grammars listed are in the format used in this document in the case of examples and defining semantic requirements. The actual internal representation in mCRL2 differs slightly, e.g. mathematic symbols are replaced by ASCII versions and keywords exist that mark the beginning or end of certain grammatical constructs.

A.1 Boolean Equation System

$$\begin{array}{rcl} BES & \rightarrow & \mathcal{E} \\ \mathcal{E} & \rightarrow & (\sigma X = \phi) \mid (\sigma X = \phi)\mathcal{E} \\ \phi & \rightarrow & \phi \Rightarrow \phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid true \mid false \mid X \\ \sigma & \rightarrow & \mu \mid \nu \end{array} \tag{A.1}$$

A.2 Data Expressions

$$\begin{array}{rcl} \psi & \rightarrow & d \mid \mathbb{N} \mid true \mid false \mid (\psi) \mid \neg\psi \mid -\psi \mid \\ & & \psi \Rightarrow \psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \psi = \psi \mid \psi \neq \psi \mid \\ & & \psi < \psi \mid \psi > \psi \mid \psi \geq \psi \mid \psi \leq \psi \mid \forall e.\psi \mid \exists e.\psi \mid \\ & & \psi + \psi \mid \psi - \psi \mid \psi * \psi \mid \psi / \psi \mid \psi\ \underline{div}\ \psi \mid \psi\ \underline{mod}\ \psi \\ \Psi & \rightarrow & \psi \mid \psi,\ \Psi \\ e & \rightarrow & d : D \mid d : D, e \end{array} \tag{A.2}$$

A.3 Parameterized Boolean Equation System

$$\begin{array}{rcl} PBES & \rightarrow & \mathcal{E} \\ \mathcal{E} & \rightarrow & (\sigma\mathcal{X} = \phi) \mid (\sigma\mathcal{X} = \phi)\mathcal{E} \\ \phi & \rightarrow & \psi \mid \forall e.\phi \mid \exists e.\phi \mid \phi \Rightarrow \phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid true \mid false \mid \mathcal{X} \\ \sigma & \rightarrow & \mu \mid \nu \\ \mathcal{X} & \rightarrow & X \mid X(\Psi) \end{array} \tag{A.3}$$

For e, ψ and Ψ, see the data expressions grammar.

A.4 mCRL2 Process Specification

$$\begin{array}{rcl} \alpha & \rightarrow & a \mid a|\alpha \\ a & \rightarrow & A \mid A(\Psi) \\ p & \rightarrow & \alpha \mid P \mid P(\Psi) \mid \delta \mid \tau \mid p + p \mid p \cdot p \mid \psi{\rightarrow}p \mid \psi{\rightarrow}p \diamond p \mid \sum_e p \\ & & p \parallel p \mid p ⫦ p \end{array} \tag{A.4}$$

For e, ψ and Ψ, see the data expressions grammar.