Instrumenting a Context-free Language Recognizer

Paulo Roberto Massa Cereda and João José Neto
Escola Politécnica, Departamento de Engenharia de Computação e Sistemas Digitais, Universidade de São Paulo, Av. Prof. Luciano Gualberto, s/n, Travessa 3, 158, CEP: 05508-900, São Paulo, SP, Brasil

Keywords: Context-free Language, Structured Pushdown Automaton, Instrumentation.

Abstract: Instrumentation plays a crucial role when building language recognizers, as collected data provide basis for achieving better performance and model improvements, thus offering a balance between time and space, as demanded by practical applications. This paper presents a simple yet functional semiautomatic approach for generating a instrumentation-aware context-free language recognizer, enhanced with hooks, from a grammar written using the Wirth syntax notation. The entire process is aided by a set of command line tools, freely available for download. We also introduce the concept of an instrumentation layer enclosing the underlying recognizer, acting as observer for each computational step and collecting data for later use.

1 INTRODUCTION

Instrumentation is the capability of monitoring or measuring performance of a device, as well as tracing information during its life cycle (Wert et al., 2015). Such metrics allow an accurate understanding of the device's inner workings and provide base for improvements on the model (Paul and Vahrenhold, 2013; Ball and Larus, 1994). In general, it is advisable to combine different metrics in order to obtain a more comprehensive representation of the device's collected data, in an attempt to reduce bias (which might cause misjudgement of the model as a whole) (Wert et al., 2015).

Language recognition devices are mechanisms capable of reading strings built from an set Σ of symbols (also known as language alphabet) and decide whether such strings are in the language they describe (Aho and Ullman, 1995). These devices play an important role in several areas, including programming languages; context-free language recognizers are widely used to design parsers (syntactic analysers), which work out the grammatical structure of strings according to a set of rules. It is highly advisable to have deterministic devices, although that is not always possible (Sebesta, 2013).

Recognizers need to be reasonably efficient, in time and space, when analysing a string. Practical applications demand a balance between these two factors (Cooper and Torczon, 2011). Hence, understanding the inner workings of such devices and particular features of the languages for which they are constructed is crucial to achieving better performance and providing model improvements (Ball and Larus, 1994). Designing instrumentation-aware recognizers allows performance monitoring and information tracing, as well as gathering potential findings about the languages themselves and their formation rules.

We present a simple yet functional semiautomatic approach for generating a instrumentation-aware context-free language recognizer from a grammar written using the Wirth syntax notation, as well as querying the recognizer and collecting instrumentation data based on a set of metrics. The entire process is aided by a set of command line tools, freely available for download.

This paper is organized as follows: Section 2 introduces the basic concepts of a context-free language recognizer, the Wirth syntax notation used to describe programming languages, and a process to automate the generation of a structured pushdown automaton given a WSN grammar. Section 3 presents the instrumentation layer, the set of metrics and its operational semantics. Conclusions are presented in Section 4.

2 BACKGROUND

In this paper, we will use a structured pushdown automaton as our recognizer for context-free languages. We aim at generating a recognizer instance from a language grammar written using the Wirth syntax notation and then instrumenting it later. The generation

Cereda, P. and Neto, J.
Instrumenting a Context-free Language Recognizer.
DOI: 10.5220/0006212002030210
In *Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS 2017) - Volume 2*, pages 203-210
ISBN: 978-989-758-248-6

will be aided by a set of command line tools written for this purpose. Before we proceed, let us formally introduce the concepts.

The structured pushdown automaton (SPA) (José Neto and Magalhães, 1981; José Neto, 1993) is a kind of pushdown automaton composed of a set of mutually recursive finite automata, also known as submachines. Unlike the traditional pushdown automaton, the stack is only used to store references to return states on each submachine call. Calls and returns consist on transferring control from one submachine to another; this special transition uses the input symbol to make a decision on which transition should be executed (the symbol is then consumed in the next transition) (José Neto, 1993; José Neto, 1994).

A structured pushdown automaton M is defined as $M = (Q, A, \Sigma, \Gamma, P, Z_0, q_0, F)$, in which Q is the set of states, A is the set of submachines, defined as follows, Σ is the automaton alphabet, corresponding to the non-empty set of input symbols, Γ is the set of stack symbols, P is the transition relation, $q_0 \in Q$ is the initial state (of the first submachine), Z_0 is a special symbol acting as an empty stack marker, and $F \subseteq Q$ is the set of accepting states (of the first submachine) (José Neto and Magalhães, 1981; José Neto, 1993).

A submachine $a_i \in A$ is defined as a traditional finite automaton $a_i = (Q_i, \Sigma_i, P_i, q_{i,0}, F_i)$, in which $Q_i \subseteq Q$ is the set of states of a_i, $\Sigma_i \subseteq \Sigma$ is the set of input symbols of a_i, $q_{i,0}$ is the entry state of a_i, $P_i \subseteq P$ is the transition relation of a_i, and $F_i \subseteq F$ is the set of return states of a_i.

The transition relation P is defined as $P \subseteq \Gamma \times Q \times \Sigma \times \Gamma \times Q$, in the form $(\gamma g, e, s\alpha) \rightarrow (\gamma g', e', \alpha)$, in which e, e' are the current and target states, respectively, s is the consumed symbol, α is the remainder of the input string, g is the current top of the stack, g' is the new top of the stack, and γ is the remainder of the stack. A configuration is an element of $Q \times \Sigma^* \times \Gamma^*$, and a relation between successive configurations $\vdash$ is defined as follows:

- *Symbol consumption:* $(q, \sigma w, uv) \vdash (p, w, xv)$, with $p, q \in Q$, $u, x \in \Gamma$, $v \in \Gamma^*$, $\sigma \in \Sigma \cup \{\varepsilon\}$, $w \in \Sigma^*$, if σ was consumed, $x = u$, e $(\gamma, q, \sigma\alpha) \rightarrow (\gamma, p, \alpha) \in P$.
- *Submachine call:* $(q, w, uv) \vdash (r, w, xv)$, with $q, r \in Q$, $u \in \Gamma$, $v, x \in \Gamma^*$, $w \in \Sigma^*$, $x = pu$, with a call to the submachine R, initial state r, return in p, and $(\gamma, q, \alpha) \rightarrow (\gamma p, r, \alpha) \in P$.
- *Submachine return:* $(q, w, uv) \vdash (p, w, v)$, with $p, q \in Q$, $u, x \in \Gamma$, $v \in \Gamma^*$, $w \in \Sigma^*$, $u = p$, with submachine return to p, and $(\gamma g, q, \alpha) \rightarrow (\gamma, g, \alpha) \in P$.

The language recognized by a structured pushdown automaton M is given by $L(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (f, \varepsilon, Z_0), f \in F\}$.

A submachine call can be graphically represented by a transition with double lines as illustrated in Figure 1. Note that, from state q_{im} of submachine a_i, execution is transferred to the submachine a_j and the address regarding the return state q_{in} is inserted into the top of the stack. In the example, the current state becomes q_{i0}, which is the initial of the submachine a_j.

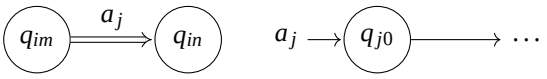

Figure 1: Example of call to the submachine a_j.

It is important to note that, as a matter of model organization, it is assumed that $a_i, a_j \in A$, $a_i = (Q_i, \Sigma_i, P_i, q_{i,0}, F_i)$, $a_j = (Q_j, \Sigma_j, P_j, q_{j,0}, F_j)$, $Q_i \cap Q_j = \emptyset$ and $P_i \cap P_j = \emptyset$, i.e. sets of states and mappings of submachines are disjoint.

Automata are devices that, based on a set of formation rules of a language, can decide whether an input string is a valid sentence, i.e. the input string is a element of the set of all sentences in that language. In the late 1970s, Wirth (Wirth, 1977) presented a metalanguage for describing programming languages, in an attempt to provide a simplified notation as alternative to existing initiatives, specially the Backus-Naur Form (BNF); such metalanguage became known as Wirth syntax notation (WSN) and has the following properties:

i) The notation shows a clear distinction between metasymbols, terminal and nonterminal symbols. Existing metasymbols are `=`, `.`, `(`, `)`, `[`, `]`, `{`, `}`, `|` and `"`. A nonterminal symbol is denoted by an identifier, i.e. one letter followed by zero or more letters and digits (as an usual variable definition in a programming language), while the terminal symbol is expressed by a string enclosed in double quotes.

ii) There is no restriction regarding the use of metasymbols as symbols of language being described. For example, the metasymbol `|` differs from terminal symbol `"|"`.

iii) The notation avoids heavy use of recursion to express simple repetitions by having a construct to express explicit iteration. Repetition is denoted by curly brackets.

iv) There is no need to use an explicit symbol to represent the empty string, such as ⟨empty⟩ in BNF or ε, because the notation already has constructs that address this situation. Optionality is expressed by square brackets.

According to Wirth (Wirth, 1977), the repetition is denoted by curly braces, i.e. `{ a }` represents ε | a | aa | aaa | ... (Kleene star). Optional elements are expressed through square brackets, i.e. `[ a ]` represents a | ε. Parentheses are used to represent grouping, i.e. `( a | b ) c` represents ac | bc. Terminal symbols are expressed enclosed in double quotes; if the double quotes appear as literal symbols, these are duplicated. Some alternative representations express literal double quotes like `"\""` instead of `""""`; Wirth's original article prefers duplicating double quotes.

The simplicity of the Wirth syntax notation allows a trivial representation of the grammar elements as internal and external transitions of the SPA (symbol consumption and submachine calls, respectively) (José Neto, 1987; José Neto et al., 1999). Given a grammar written in WSN, we can use the SPA presented in Figure 2 in order to obtain a resulting SPA that recognizes sentences from the language expressed in the provided grammar (José Neto, 1987; José Neto et al., 1999; Cereda and José Neto, 2015). Semantic actions associated with transitions are described in Figure 3.

The resulting SPA is potentially nondeterministic; however, as each submachine is in itself a finite automaton, the automaton could be translated to an equivalent deterministic SPA using classic subset construction algorithms (Cooper and Torczon, 2011; Sebesta, 2013). Also, each submachine could be reduced to an equivalent automaton with a minimum number of states through minimization (Hopcroft, 1971).

We will use a command line tool named `wsn2spa`[1] in order to automate the SPA generation from a grammar written in WSN; there are options for deterministic translation and state minimization as well. The tool is written in Java and it is released under GPLv3 (the GNU General Public License 3.0). The default output is a DOT (plain text graph description language) file, but we are also interested in the secondary format, a YAML (human-friendly data serialization standard) file, which provides a textual, structural representation of the resulting SPA. We will discuss the usage later on, in the next section.

3 INSTRUMENTING A RECOGNIZER

Consider the automation flow presented in Figure 4. From a grammar, written in WSN, representing arithmetic expressions (for simplicity purposes, we are only considering addition and nested parentheses), `wsn2spa` generates a SPA spec. The language is clearly context-free; valid sentences include `a`, `a + a`, `(a + a)`, `a + (a + a)`, and so on. The graphical representation of this specific SPA spec is presented in Figure 5.

Note that the call to `wsn2spa` shown in Figure 4 included two optional flags, `-c` and `-m`. As the output indicates, the generated SPA had the submachine *AE* translated to its equivalent minimized deterministic finite automaton. The tool also generated a DOT file representing the submachine *AE* (and each additional operation applied to it); the file can be compiled with the `dot` command (from GraphViz). If the SPA had more submachines, the tool would generate a set of DOT and YAML files representing each submachine.

Once we have the SPA spec (possibly comprised of individual submachine specs), we can use another helper tool, named `spa2run`[2], in order to submit string queries to the automaton and check whether they are valid sentences in the language the SPA recognizes. The tool is also written in Java and it is released under GPLv3, just as `wsn2spa`. The input takes a list of submachine specs written in YAML (being the first item in the list the main submachine); once this list is provided, the tool generates an on-the-fly executable code and grants a shell session in order to query the automaton. The user can abort the session at any time by pressing a certain combination of keys or using the reserved keyword `:quit` as input string. Figure 6 shows `spa2run` in action, as it instantiates the SPA spec generated in Figure 4 into a proper automaton and allows querying it.

Now that we have means of describing a context-free language through WSN, generate its corresponding recognizer (namely, a SPA) and query an automaton instance to check whether an input string is a valid sentence of that language, we are able to go further and instrument the recognizer. But first, let us formally introduce the operational semantics of our instrumentation.

Let us define a set $B = \{b \mid b\colon P \mapsto \mathbb{R}\}$ of instrumentation metrics, i.e. a set of functions that takes an element of the transition relation and returns a real value. This approach allows us to simultaneously apply several metrics to the very same recognition instance.

In order to keep track of each metric, the instrumentation layer (presented in Figure 7) provides a list V of real variables, $|V| = |B|$, such that each variable $v_i \in V$ is associated with a function $b \in B$. At first, $\forall v_i \in V, v_i \leftarrow 0$. For each computational step in the

[1] Official repository: https://goo.gl/pULqpm

[2] Official repository: https://goo.gl/MvCnQs

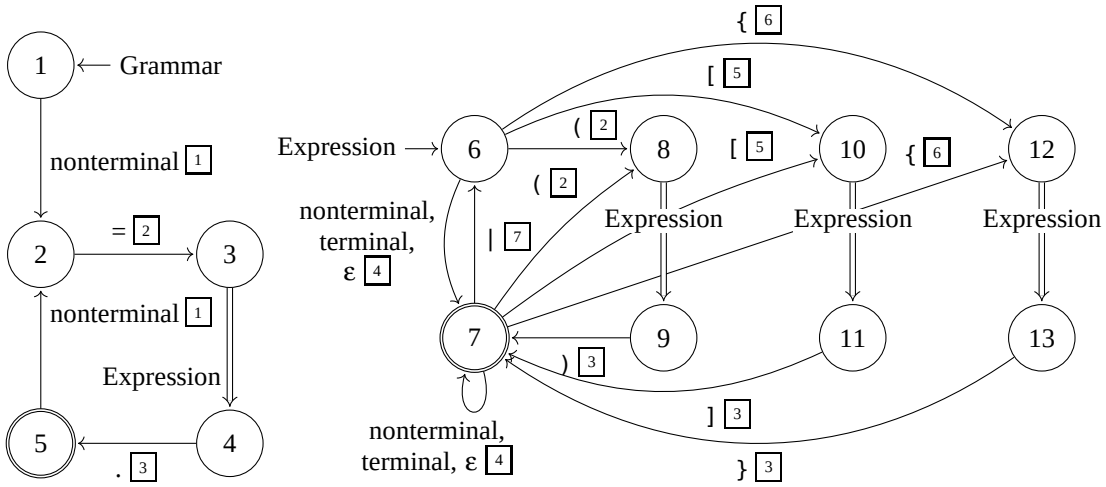

Figure 2: SPA that generates another SPA given a grammar written in WSN.

Semantic action 1

```
stack.empty(); current := 0; counter := 1
```

Semantic action 2

```
stack.push(pair(current, counter))
counter := counter + 1
```

Semantic action 3

```
stack.empty()
transitions.add(transition(current,
     epsilon, stack.top().second())
current := stack.top().second()
stack.pop()
```

Semantic action 4

```
transitions.add(transition(current,
     token, counter))
counter := counter
```

Semantic action 5

```
transitions.add(transition(current,
     epsilon, counter))
stack.push(pair(current, counter))
counter := counter + 1
```

Semantic action 6

```
transitions.add(transition(current,
     epsilon, counter))
current := counter
stack.push(pair(counter, counter))
counter := counter + 1
```

Semantic action 7

```
transitions.add(transition(current,
     epsilon, stack.top().second()))
current := stack.top().first()
```

Figure 3: Semantic actions associated with transitions of the SPA from Figure 2.

Grammar, written in WSN

```
AE = ( "a" | "(" AE ")" )
     { "+" ( "a" | "(" AE ")" )  } .
```

Running the tool...

```
$ java -jar wsn2spa.jar grammar.txt \
-o dot%s.dot -y spec%s.yaml -c -m
[lines omitted]
- Submachines translated to DFA's.
- State minimization applied.
```

YAML spec for the main submachine

```
name: AE
initial: 0
accepting: [1]
transitions:
- { from: 0, symbol: a, to: 1 }
- { from: 0, symbol: (, to: 2 }
- { from: 1, symbol: +, to: 0 }
- { from: 2, symbol: AE (call), to: 3 }
- { from: 3, symbol: ), to: 1 }
```

Figure 4: Automation flow from a grammar representing simple arithmetic expressions (only addition and nested parentheses), written in WSN, to the corresponding SPA spec.

recognition process of a string w, the current matched transition $p \in P$ is measured through each function $b_i \in B$ and the result is added to its corresponding variable $v_i \in V$, such that $v_i \leftarrow v_i + b_i(p)$. It is important to observe that we require every function in B to be total, thus $\forall p \in P$, if $b_i(p)$ is not explicitly defined, $b_i(p) = 0$ by definition.

As an example, consider the SPA presented in Fig-

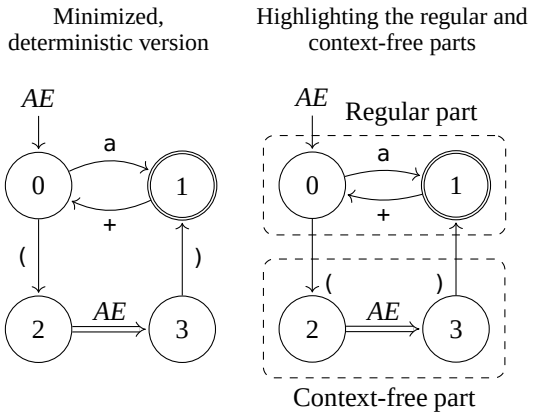

Figure 5: Graphical representation of the SPA spec from Figure 4.

ure 8. For simplicity's sake, the stack was omitted. This particular automaton recognizes sentences from a regular language L, defined as $L = \{w \in \{a,b\}^* \mid w = abb^*a(bb^*a)^*\}$. Now, let us introduce a set $B = \{b_1, b_2\}$ of instrumentation metrics such that:

- b_1 is a total function, $b_1 \colon P \mapsto \mathbb{R}$, in which $b_1(\langle 0,a,1\rangle) = 1.0$, $b_1(\langle 1,b,2\rangle) = 3.0$, $b_1(\langle 2,b,2\rangle) = 2.0$, $b_1(\langle 2,a,3\rangle) = 4.0$, $b_1(\langle 3,b,2\rangle) = 6.0$.
- b_2 is a total function, $b_2 \colon P \mapsto \mathbb{R}$, in which $b_2(\langle 0,a,1\rangle) = 0.34$, $b_2(\langle 1,b,2\rangle) = 1.42$, $b_2(\langle 2,b,2\rangle) = 3.87$, $b_2(\langle 2,a,3\rangle) = 3.21$, $b_2(\langle 3,b,2\rangle) = 5.16$.

Given a string $w = abbaba$, the instrumentation layer collects each measurement during the recognition process and updates the variables accordingly. When the automaton effectively stops, as there are no valid transitions to apply given the current symbol (none, actually), the variables hold the instrumentation results, namely $v_1 = 20.0$ and $v_2 = 17.21$ (values corresponding to the sum of the application of a function $b_i \in B$ to each matched transition $p \in P$ when recognizing w).

In this paper, the set B of instrumentation metrics is solely composed of functions that take an element $p \in P$ and return a real value; also, being $M = \langle m_1 \dots m_n \rangle$, $\forall m_i \in M, m_i \in P$, the sequence of all matched transitions during the recognition process of a string w, the instrumentation results are stored in a list V of variables, such that $\forall v_i \in V, v_i = \sum_{j=1}^{|M|} b_i(m_j)$, i.e. each variable holds the sum of its corresponding instrumentation function taking each element in the list of matched transitions. However, these concepts could be generalized in order to cover other domains, as well as applying complex data manipulation and computation.

Now that the concept has been formally introduced, let us see how one can use `spa2run` to instrument a SPA recognizer and collect metrics accordingly. But first, we need to add hooks to our automaton spec in order to make it instrumentation-aware, so the layer can detect whether a computational step should be measured.

As to reduce verbosity, each transition $p \in P$ is labeled with a positive integer $k \in \mathbb{Z}, k > 0$, acting as a unique identifier and hook regarding p. This approach favours the design principle of separation of concerns, as it provides a straightforward interface such that each potential layer enclosing an underlying recognizer and acting as an observer for each computational step addresses a separate processing based on hooks. Figure 9 presents the SPA from Figure 5 enhanced with hooks, graphically represented as framed numbers. Note that the notation for representing hooks was informally introduced in Figure 2, in which transitions are associated with semantic actions; this is no coincidence, as hooks act an interface to layers, regardless of their underlying processing. We will discuss this later on, in the next section.

According to Figure 9, every transition $p \in P$ contains an associated positive integer $i, i \in \mathbb{Z}$, acting as a unique identifier and hook. Now, let us introduce a set $B = \{b_1\}$ of instrumentation metrics, such that b_1 is a total function, $b_1 \colon P \mapsto \mathbb{R}$, in which $b_1(\langle 0,\mathtt{a},\gamma,1,\varepsilon,\gamma\rangle) = 2.0$, $b_1(\langle 1,\mathtt{+},\gamma,0,\varepsilon,\gamma\rangle) = 6.0$, $b_1(\langle 0,\mathtt{(},\gamma,2,\varepsilon,\gamma\rangle) = 7.0$, and $b_1(\langle 3,\mathtt{)},\gamma,1,\varepsilon,\gamma\rangle) = 5.0$. Note that we deliberately omitted the submachine call, namely $2 \to 3$, as such transition will not be measured; hence, by definition, $\forall p' \in P$, p' is a submachine call, $b_1(p') = 0$.

The `spa2run` tool uses a lookup function ψ, such that $\psi \colon \mathbb{Z} \mapsto P$, i.e. the function maps identifiers to their corresponding transitions. Given the SPA from Figure 9 and the set B of instrumentation metrics previously defined, $\psi(1) = \langle 0,\mathtt{a},\gamma,1,\varepsilon,\gamma\rangle$, $\psi(2) = \langle 1,\mathtt{+},\gamma,0,\varepsilon,\gamma\rangle$, $\psi(3) = \langle 0,\mathtt{(},\gamma,2,\varepsilon,\gamma\rangle$, $\psi(4) = \langle 3,\mathtt{)},\gamma,1,\varepsilon,\gamma\rangle$, and $\psi(5) = \langle 2,\sigma,\gamma,0,\sigma,3\gamma\rangle$. The tool combines the provided set B of instrumentation metrics with its lookup function in order to query values based on transition identifiers, e.g. $b_1(\psi(1)) = 2.0$, which is much more understandable from the user's point of view.

As the next step, we need to update the SPA spec from Figure 4 in order to make it instrumentation-aware. The update is straightforward, as it suffices to include an `identifier` key to each transition definition, matching the numbering scheme from Figure 9. An excerpt of the new spec is presented in Figure 10.

Once the SPA is updated with the corresponding transition identifiers (as seen in Figure 10), `spa2run`

Running the tool...

```
$ java -jar spa2run.jar \
specEA.yaml
[lines omitted]
Starting shell, please wait...
(press CTRL+C or type ':quit'
to exit the application)
```

Queries

```
[1] query> (a+(a+a+a))
[1] result> true (deterministic)

[2] query> (a+(a+a)
[2] result> false (deterministic)
```

Figure 6: The `spa2run` tool instantiates the SPA spec generated from `wsn2spa` (Figure 4) as a proper automaton and provides a shell session for queries.

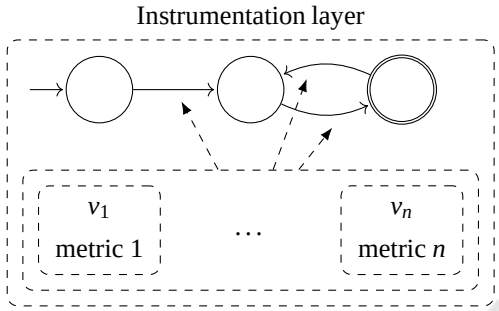

Figure 7: The instrumentation layer, enclosing an underlying recognizer. Note that the layer acts as an observer for each computational step and updates the list of variables according to their corresponding instrumentation metrics.

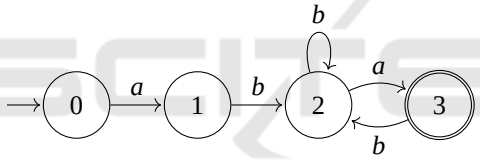

Figure 8: SPA that recognizes sentences from $L = \{w \in \{a,b\}^* \mid w = abb^*a(bb^*a)^*\}$.

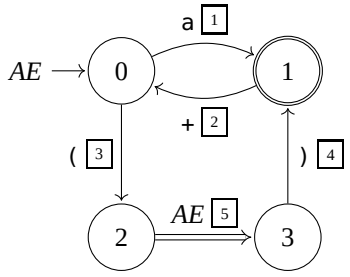

Figure 9: SPA from Figure 5 enhanced with hooks, graphically represented as framed numbers.

can now take a list of metrics specs, using the `-i` flag, and instrument the recognizer accordingly. Figure 11 illustrates the instrumentation metric $b_1 \in B$, previously defined and codified as a spec in the YAML format, as well as the new instrumentation-aware shell session. Note that the metric spec deliberately ommits the submachine call, as it will not be measured.

Consider a second instrumentation metric b_2 added to the set B of instrumentation metrics defined previously, such that $b_2 \in B, B = \{b_1, b_2\}$,

```
transitions:
- { from: 0, symbol: a, to: 1, identifier: 1 }
- { from: 0, symbol: (, to: 2, identifier: 3 }
- { from: 1, symbol: +, to: 0, identifier: 2 }
- { from: 2, symbol: AE (call),
    to: 3, identifier: 5 }
- { from: 3, symbol: ), to: 1, identifier: 4 }
```

Figure 10: Excerpt of the new YAML spec for the SPA from Figure 9, including the transition identifiers (associated positive integers) in order to make it instrumentation-aware.

and b_2 is a total function, $b_2 \colon P \mapsto \mathbb{R}$, in which $b_2(\langle 0, \mathtt{a}, \gamma, 1, \varepsilon, \gamma\rangle) = 3.2$, $b_2(\langle 1, +, \gamma, 0, \varepsilon, \gamma\rangle) = 1.78$, $b_2(\langle 0, (, \gamma, 2, \varepsilon, \gamma\rangle) = 5.57$, and $b_2(\langle 3,), \gamma, 1, \varepsilon, \gamma\rangle) = 4.23$. As in b_1, note that we deliberately omitted the submachine call, namely $2 \to 3$, as such transition will not be measured; hence, by definition, $\forall p' \in P$, p' is a submachine call, $b_2(p') = 0$. Figure 12 shows the corresponding metric spec codified in the YAML format, as well as queries from `spa2run` using the set B of instrumentation metrics.

Instrumentation metrics can be used to evaluate the recognizer's behaviour over time and study characteristics of the language itself, based on individual analysis of a group of strings. For instance, consider the SPA from Figure 5, the set B of instrumentation metrics, previosuly defined, L_i is defined as $L_i = \{w \in L \mid |w| = i\}$, i.e. a subset of language L in which all valid strings have length i, and $\forall i \in \mathbb{Z}, i \bmod 2 = 0, L_i = \emptyset$, i.e. only strings with odd lengths are valid (the subset of valid strings with even lengths is empty); let us write an evaluation test and instrument every string $w \in L_i, 3 \leq i \leq 11, i \bmod 2 \neq 0$, such that $x_{i,j} = \sum_{w \in L_i}^{L_i} v_j / |L_i|$, with $1 \leq j \leq |B|$, i.e. $x_{i,j}$ holds the arithmetic mean of the instrumentation results from v_j applied to all valid strings of length i, and v_j is associated to the sum of the application of a function $b_j \in B$ to each matched transition $p \in P$ when recognizing $w \in L_i$. We applied a string generator based on the WSN grammar of L in order to generate all valid strings $w \in L_i$ and then used `spa2run` as an instrumentation interface to the SPA. The obtained results are presented in Figure 13.

According to Figure 13, the instrumentation results grow proportionally to the length of a string

Metric codified as a YAML spec

```
name: B1
mapping:
- { identifier: 1, value: 2.0 }
- { identifier: 2, value: 6.0 }
- { identifier: 3, value: 7.0 }
- { identifier: 4, value: 5.0 }
```

Running the tool...

```
$ java -jar spa2run.jar \
specEA.yaml -i b1.yaml
```

Queries

```
[1] query> (a+(a+a+a))
[1] result> true (deterministic)
    B1: 50.0

[2] query> ((a))
[2] result> true (deterministic)
    B1: 26.0
```

Figure 11: Instrumentation metric b_1, codified as a spec in the YAML format, and a instrumentation-aware shell session from `spa2run`, using the new SPA spec from Figure 10. Note that the instrumentation metric used by `spa2run` relies on identifiers instead of transitions, as to reduce verbosity.

Metric codified as a YAML spec

```
name: B2
mapping:
- { identifier: 1, value: 3.20 }
- { identifier: 2, value: 1.78 }
- { identifier: 3, value: 5.57 }
- { identifier: 4, value: 4.23 }
```

Running the tool...

```
$ java -jar spa2run.jar \
specEA.yaml -i b1.yaml b2.yaml
```

Queries

```
[1] query> (a+(a+a+a))
[1] result> true (deterministic)
    B1: 50.0 B2: 37.74

[2] query> a+(a+a)
[2] result> true (deterministic)
    B1: 30.0 B2: 22.96
```

Figure 12: Instrumentation metric b_2, codified as a spec in the YAML format, and a instrumentation-aware shell session from `spa2run`, using the new SPA spec from Figure 10 and the set $B = \{b_1, b_2\}$ of instrumentation metrics.

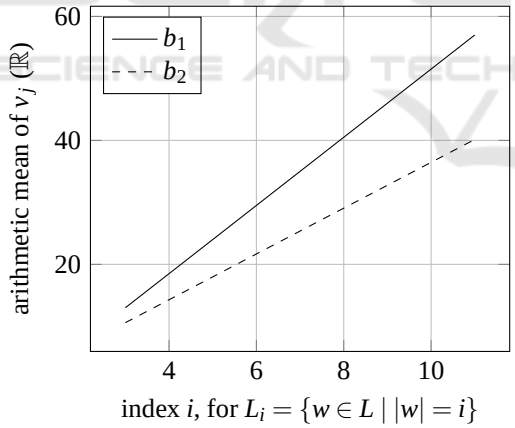

Figure 13: Arithmetic mean of the instrumentation results from v_j applied to all valid strings of length $i, 3 \leq i \leq 11$, with v_j associated to the sum of the application of a function $b_j \in B$ to each matched transition $p \in P$ when recognizing $w \in L_i$.

$w \in L$, given the formation rules of this particular language, its corresponding recognition device and set of instrumentation metrics. These clues might provide subsidies for further studies, e.g. improving sections of a syntactic analyser in order to avoid costly paths (in our case, balanced parentheses could be checked with a counter instead of a submachine call and stack operations, as they do not influence the result in an arithmetic expression only containing addition operations).

4 CONCLUSIONS

This paper presented a semiautomatic approach for generating a instrumentation-aware context-free language recognizer, enhanced with hooks, from a grammar written using the Wirth syntax notation. The entire process was aided by a set of command line tools, freely available for download. We also introduced the concept of an instrumentation layer enclosing the underlying recognizer, acting as observer for each computational step and collecting data for later use.

Observe that the unique identifiers associated to transitions can be extended beyond instrumentation, as they provide a convenient feature for appending metrics, data structures and semantic actions. Figure 2 illustrates how hooks are associated with semantic actions, in this case, generating another SPA given a grammar written in WSN. Semantic actions are triggered whenever their corresponding transitions match the device's current configuration.

We are working on a syntax for specifying com-

plex operations in `spa2run` using scripting languages running on the Java Virtual Machine, such as Groovy, Scala and BeanShell. To this end, the instrumentation layer is being generalized in order to natively accommodate more elements, such as data structures and semantic actions. Furthermore, there is an ongoing research on instrumenting context-sensitive language recognizers, namely adaptive automata (José Neto, 1994; José Neto, 2001) (rule-driven devices exploiting self-modification by adding an adaptive layer on top of the underlying rule set); we are using a library named AA4J (Cereda and José Neto, 2016), which allows an straightforward implementation of adaptive automata, and extending it with hooks for later instrumentation. Preliminary results look promising, although there are challenges regarding the computational costs when modifying the underlying device (which is, by definition, a SPA enclosed by an adaptive layer).

Instrumenting language recognizers allows a better understanding of the inner workings of such devices as well as particular features of the languages for which they are constructed. Collected data from instrumentation provide basis for achieving better performance and model improvements, thus offering a balance between time and space, as demanded by practical applications. Besides, the use of an instrumentation layer enclosing an underlying recognizer provides a framework generic enough to cover several domains.

REFERENCES

Aho, A. V. and Ullman, J. D. (1995). *Foundations of Computer Science*. W. H. Freeman and Company.

Ball, T. and Larus, J. R. (1994). Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360.

Cereda, P. R. M. and José Neto, J. (2015). Um arcabouço para extensibilidade em linguagens de programação. In *Memórias do IX Workshop de Tecnologia Adaptativa – WTA 2015*, pages 18–28, São Paulo.

Cereda, P. R. M. and José Neto, J. (2016). AA4J: uma biblioteca para implementação de autômatos adaptativos. In *Memórias do X Workshop de Tecnologia Adaptativa – WTA 2016*, pages 16–26.

Cooper, K. and Torczon, L. (2011). *Engineering a compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Hopcroft, J. E. (1971). An $n \log n$ algorithm for minimizing states in a finite automaton. In *Proceedings of the International Symposium on the Theory of Machines and Computations*, pages 189–196, Haifa, Israel. Academic Press.

José Neto, J. (1987). *Introdução à compilação*. Engenharia de Computação. LTC – Livros Técnicos e Científicos.

José Neto, J. (1993). *Contribuições à metodologia de construção de compiladores*. Postdoctoral thesis, Escola Politécnica da Universidade de São Paulo, São Paulo.

José Neto, J. (1994). Adaptive automata for context - sensitive languages. *SIGPLAN Notices*, 29(9):115–124.

José Neto, J. (2001). Adaptive rule-driven devices: general formulation and case study. In *International Conference on Implementation and Application of Automata*.

José Neto, J. and Magalhães, M. E. S. (1981). Reconhecedores sintáticos: Uma alternativa didática para uso em cursos de engenharia. In *XIV Congresso Nacional de Informática*, pages 171–181.

José Neto, J., Pariente, C. B., and Leonardi, F. (1999). Compiler construction: a pedagogical approach. In *Proceedings of the V International Congress on Informatic Engineering – ICIE 99*, Buenos Aires, Argentina.

Paul, W. and Vahrenhold, J. (2013). Hunting high and low: Instruments to detect misconceptions related to algorithms and data structures. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, pages 29–34, New York, NY, USA. ACM.

Sebesta, R. W. (2013). *Concepts of Programming Languages*. Pearson, 10 edition.

Wert, A., Schulz, H., Heger, C., and Farahbod, R. (2015). Generic instrumentation and monitoring description for software performance evaluation. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 203–206, New York, NY, USA. ACM.

Wirth, N. (1977). What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 10(11):822–823.