# AASL: Gramática para uma Linguagem de Especificação de Autômatos Adaptativos

P. Cereda, I. S. Vega

Outubro de 2016

## Sumário



# Introdução

Este relatório oferece um projeto alternativo para a linguagem de especificação de autômatos adaptativos criada por Cereda https://github.com/cereda/xml2aa. No momento, parece ter um funcionamento equivalente ao daquele repositório sob o ponto de vista do objeto da classe `AdaptiveAutomaton`. A intenção desta revisão é a de facilitar o estudo de diferentes estruturas gramaticais da linguagem por meio de uma notação baseada em regras de produção.

A ferramenta de geração de compiladores ANTLR4 foi utilizada, uma vez que preserva a possibilidade de se realizar a análise semântica na linguagem Java, além de oferecer suporte a decisões arquiteturais com um grau menor de acoplamento em relação as do projeto original.

Os seguintes exemplos reproduzem aqueles do projeto original, mas usando a linguagem proposta nesta revisão. Inicialmente, um exemplo de autômato finito. Graficamente representado na Fig 1, encontra-se um autômato constituído por três estados, sendo $q_0$ o estado inicial e $q_2$ o estado de aceitação.

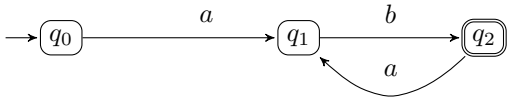


Figura 1: Exemplo de um autômato finito

```
<<af.aal>>=
transitions
    from 0 symbol a to 1
    from 1 symbol b to 2
    from 2 symbol a to 1

submachines
    submachine M main
        state 0 start
        state 1
        state 2 accepting
```

Em seguida, um autômato de pilha estruturado [#fig:exemplo-20].

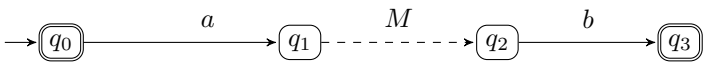


Figura 2: Exemplo de um autômato finito

```
<<ape.aal>>=
transitions
    from 0 symbol a to 1
    from 1 call M to 2
    from 2 symbol b to 3

submachines
    submachine M main
        state 0 start accepting
        state 1
        state 2
        state 3 accepting
```

Finalmente, um autômato adaptativo:

```
<<aa.aal>>=
transitions
    from 0 symbol a to 1
    from 1 symbol b to 2
    from 2 symbol c to 3
    from 1 symbol a to 3
```

```
        postAdaptiveFunction A ( 2, 3 )

submachines
    submachine M main
        state 0 start
        state 1
        state 2
        state 3 accepting

actions
    adaptiveAction A ( p1, p2 )
        local ?x, ?y
        generator g1*, g2*
        action query ?x b p1
        action remove ?x b p1
        action query ?y c p2
        action remove ?y c p2
        action remove 1 a 1
            postAdaptiveFunction A (p1, p2)
        action add ?x b g1*
        action add g1* b p1
        action add ?y c g2*
        action add g2* c p2
        action add 1 a 1
            postAdaptiveFunction A (g1*, g2*)
```

# Tradução Dirigida por Sintaxe

O processamento da linguagem de especificação envolve a gramática descrita a seguir. A linguagem **AASL**possui a seguinte estrutura gramatical de alto nível:

```
<<AASL.g4>>=
grammar AASL;
    adaptiveAutomaton: ( transitions | submachines | actions )* ;

    <<AASL::transições>>
    <<AASL::submáquinas>>
    <<AASL::ações>>

    <<AASL::regras léxicas>>
```

A especificação de um autômato adaptativo envolve uma série de transições, submáquinas e ações adaptativas. A última parte da gramática refere-se às regras do analisador léxico:

```
<<AASL::regras léxicas>>=
ID: ( CAR | DIG )+ ;
DIG: [0-9] ;
CAR: [a-zA-Z] ;
```

```
BR : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+ -> skip ;
```

## Objeto Semântico

A análise semântica é dirigida pela estrutura sintática. Mais especificamente, um percurso na árvore sintática construída pelo processamento da especificação de entrada. O diagrama de classes UML mostrado na figura a seguir ilustra a arquitetura da ferramenta de processamento da linguagem **AASL**:

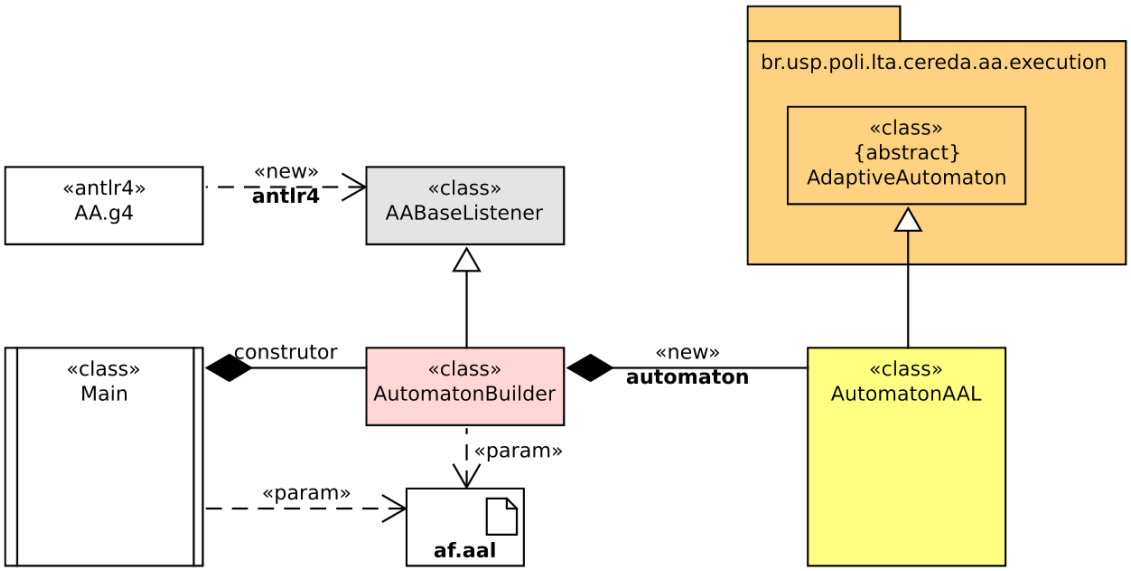


Figura 3:

O processo principal constrói um objeto da classe `ParseTreeWalker`, responsável por conhecer a gramática da linguagem **AASL**. Este objeto recebe um `AutomatonBuilder` e um contexto. Ao percorrer uma árvore sintática, ele chama o `AutomatonBuilder` para que este, gradativamete, construa um objeto semântico da classe `AutomatonAAL`, um caso especial de comportamento oferecido pela biblioteca `br.usp.poli.lta.cereda.aa.execution.AdaptiveAutomaton`. No diagrama, representa-se a entrada sugestiva `af.aal`, correspondente ao primeiro exemplo de autômato finito.

Eis o programa principal:

```
<<Main.java>>=
package gram;
<<AASL::módulos importados>>
public class Main {
    public static void main(String[] args ) {
        Main main = new Main();
        main.processar( "test/af.aal" );
    }
```

```
    <<Main::processamento principal>>
}
```

A sua execução depende dos seguintes módulos:

```
<<AASL::módulos importados>>=
import gram.antlr4.AASLLexer;
import gram.antlr4.AASLParser;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import org.antlr.v4.runtime.ANTLRInputStream;
import org.antlr.v4.runtime.CommonTokenStream;
import org.antlr.v4.runtime.tree.ParseTreeWalker;
```

Essencialmente, o processamento principal instancia os elementos do arcabouço ANTLR4 e configura os objetos do processador **AASL** para que seja construído o objeto semântico:

```
<<Main::processamento principal>>=
private void processar(String espec) {
    InputStream is = null;
    try {
        is = new FileInputStream(espec);
        ANTLRInputStream input = new ANTLRInputStream(is);
        AASLLexer lexer = new AASLLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        AASLParser parser = new AASLParser(tokens);
        AASLParser.AdaptiveAutomatonContext contexto = parser.adaptiveAutomaton();
        ParseTreeWalker analisador = new ParseTreeWalker();
        AutomatonBuilder construtor = new AutomatonBuilder();
        analisador.walk(construtor, contexto);
        System.out.println(construtor.automaton);
    } catch (IOException ex) {
        System.out.println("** FATAL! Problema de acesso ao arquivo: " + espec);
        System.exit(0);
    } finally {
        try {
            is.close();
        } catch (IOException ex) {
            System.out.println("** FATAL! Problema ao fechar o arquivo: " + espec);
            System.exit(0);
        }
    }
}
```

O resultado final do processamento de uma especificação de autômato adaptativo é um objeto da classe `AutomatonAAL`:

```
<<AutomatonAAL.java>>=
package gram;
import br.usp.poli.lta.cereda.aa.execution.AdaptiveAutomaton;
```

```
import br.usp.poli.lta.cereda.aa.model.Submachine;
import br.usp.poli.lta.cereda.aa.model.Transition;
public class AutomatonAAL extends AdaptiveAutomaton {

    public void add(Transition t) {
        transitions.add(t);
    }

    @Override
    public void setup() {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    void addSubMachine(Submachine nova) {
        submachines.add(nova);
    }
}
```

## Construtor de Autômatos

O construtor de autômatos é acionado pelo `ParseTreeWalker` para que ele produza um objeto semântico correspondente à especificação fornecida como entrada:

```
<<AutomatonBuilder.java>>=
package gram;
<<AutomatonBuilder::módulos importados>>
public class AutomatonBuilder extends AASLBaseListener {
    <<AutomatonBuilder::construtor>>
    <<AutomatonBuilder::encerramento do processo de construção>>
    <<AutomatonBuilder::processamento de transições>>
    <<AutomatonBuilder::processamento de submáquinas>>
    <<AutomatonBuilder::processamento de funções adaptativas>>
}
```

A análise semântica importa diversos elementos das bibliotecas `br.usp.poli.lta.cereda.aa`, ANTLR4 e Java:

```
<<AutomatonBuilder::módulos importados>>=
import br.usp.poli.lta.cereda.aa.examples.ExampleState;
import br.usp.poli.lta.cereda.aa.examples.ExampleSymbol;
import br.usp.poli.lta.cereda.aa.model.State;
import br.usp.poli.lta.cereda.aa.model.Submachine;
import br.usp.poli.lta.cereda.aa.model.Transition;
import br.usp.poli.lta.cereda.xml2aa.model.ListAction;
import gram.antlr4.AASLBaseListener;
import gram.antlr4.AASLParser;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Set;
```

Imediatamente após instanciado, o construtor referencia um objeto da classe `AutomatonAAL`, que será gradualmente especializado, de acordo com a especificação fornecida para processamento:

```
<<AutomatonBuilder::construtor>>=
AutomatonBuilder() {
    super();
    automaton = new AutomatonAAL();
}
public AutomatonAAL automaton;
Set<String> states = new HashSet<>(); // mantive, mas acho que não precisa...
```

No final do processamento, "método gerador de estados é atualizado para conter o próximo inteiro do conjunto de estados inteiros" (Cereda):

```
<<AutomatonBuilder::encerramento do processo de construção>>=
@Override
public void exitAdaptiveAutomaton(AASLParser.AdaptiveAutomatonContext ctx) {
    ListAction.setCounter(states.stream().mapToInt(Integer::parseInt).
        max().getAsInt() + 1);
}
```

# Especificação de Transições

A especificação de uma transição envolve o estado de origem e o de destino. O estímulo para disparo da transição pode ser representado por um símbolo ou por uma chamada de submáquina. Além disso, pode-ser vincular uma sequência de chamadas de funções adaptativas *before* e *after*:

```
<<AASL::transições>>=
transitions: 'transitions' transition+;
transition: from stimuli to transPreAdaptive* transPostAdaptive* ;
    from: 'from' ID ;
    to: 'to' ID ;
    stimuli: simbolo | chamada ;
        simbolo: 'symbol' ID ;
        chamada: 'call' ID ;
    transPreAdaptive: 'preAdaptiveFunction' ID '(' argLst ')' ;
    transPostAdaptive: 'postAdaptiveFunction' ID '(' argLst ')' ;
    argLst : arg ( ',' arg )* ;
```

## Processamento de Transições

```
<<AutomatonBuilder::processamento de transições>>=
Transition t; // objeto que representa uma transição sendo construída

@Override
public void enterTransition(AASParser.TransitionContext ctx) {
```

```
    t = new Transition();
    String from = ctx.from().ID().getText();
    String to = ctx.to().ID().getText();
    t.setSourceState(new ExampleState(from));
    t.setTargetState(new ExampleState(to));
}

@Override
public void exitTransition(AASParser.TransitionContext ctx) {
    automaton.add(t);
}

<<AutomatonBuilder::transições::estados de origem e de destino>>
<<AutomatonBuilder::transições::estímulo da transição>>
<<AutomatonBuilder::transições::chamadas adaptativas>>
<<AutomatonBuilder::transições::mecanismo para processar argumentos>>
```

## Estados de Origem e Destino

```
<<AutomatonBuilder::transições::estados de origem e de destino>>=
@Override
public void exitFrom(AASParser.FromContext ctx) {
    states.add(ctx.ID().getText());
}

@Override
public void exitTo(AASParser.ToContext ctx) {
    states.add(ctx.ID().getText());
}
```

## Estímulo da Transição

```
<<AutomatonBuilder::transições::estímulo da transição>>=

@Override
public void exitSimbolo(AASParser.SimboloContext ctx) {
    String nome = ctx.ID().getText();
    t.setSymbol(new ExampleSymbol(nome));
}

@Override
public void exitChamada(AASParser.ChamadaContext ctx) {
    String nome = ctx.ID().getText();
    t.setSubmachineCall(nome);
}
```

### Chamadas Adaptativas em Transições

```
<<AutomatonBuilder::transições::chamadas adaptativas>>=
@Override
public void exitTransPreAdaptive(AASParser.TransPreAdaptiveContext ctx) {
    String nome = ctx.ID().getText();
    t.setPriorActionCall(nome);
    argLst = new ArrayList<>();
    t.setPriorActionArguments(argLst.toArray());
    argLst = null;
}

@Override
public void enterTransPostAdaptive(AASParser.TransPostAdaptiveContext ctx) {
    String nome = ctx.ID().getText();
    t.setPostActionCall(nome);
    argLst = new ArrayList<>();
}

@Override
public void exitTransPostAdaptive(AASParser.TransPostAdaptiveContext ctx) {
    t.setPostActionArguments(argLst.toArray());
    argLst = null;
}
```

### Argumentos de Chamadas Adaptativas em Transições

```
<<AutomatonBuilder::transições::mecanismo para processar argumentos>>=
@Override
public void exitArg(AASParser.ArgContext ctx) {
    String param = ctx.ID().getText();
    argLst.add(param);
}
ArrayList<String> argLst;
```

## Especificação de Submáquinas

Submáquinas, por sua vez, definem um espaço de estados internos, inicial ou terminais (de aceitação). Uma delas, também, pode ser marcada como a submáquina principal:

```
<<AASL::submáquinas>>=
submachines: 'submachines' submachine+ ;
submachine: 'submachine' ID main state+ ;
    main: 'main'? ;
    state: 'state' ID inicial? terminal? ;
```

```
        inicial: 'start' ;
        terminal: 'accepting' ;
```

## Processamento de Submáquinas

```
<<AutomatonBuilder::processamento de submáquinas>>=
@Override
public void enterSubmachine(AASParser.SubmachineContext ctx) {

    subMaquinaEstadoLst = new HashSet<>();
    subMaquinaAceitacaoLst = new HashSet<>();
    subMaquinaInicial = null;
}

@Override
public void exitSubmachine(AASParser.SubmachineContext ctx) {
    String nome = ctx.ID().getText();
    Submachine nova = new Submachine(nome,
        subMaquinaEstadoLst, subMaquinaInicial, subMaquinaAceitacaoLst);
    automaton.addSubMachine(nova);
    if (ctx.main() != null) {
        automaton.setMainSubmachine(nome);
    }
}

@Override
public void enterState(AASParser.StateContext ctx) {
    estadoCorrente = new ExampleState(ctx.ID().getText());
    subMaquinaEstadoLst.add(estadoCorrente);
}
State estadoCorrente;
Set<State> subMaquinaEstadoLst;
State subMaquinaInicial;
Set<State> subMaquinaAceitacaoLst;

@Override
public void enterInicial(AASParser.InicialContext ctx) {
    subMaquinaInicial = estadoCorrente;
}

@Override
public void enterTerminal(AASParser.TerminalContext ctx) {
    subMaquinaAceitacaoLst.add(estadoCorrente);
}
```

# Especificação de Funções Adaptativas

Finalmente, as funções adaptativas. Cada uma delas possui uma estrutura sintática mais elaborada do que os constructos anteriores. Deve ser possível parametrizá-las e, no seu escopo local, adicionar variáveis locais, geradores e ações adaptativas. Estas últimas admitem chamadas de outras funções adaptativas:

```
<<AASL::ações>>=
actions: 'actions' adaptiveAction+ ;
adaptiveAction: 'adaptiveAction' ID '(' paramLst ')'
                    local?
                    generator?
                    action* ;

    paramLst : param ( ',' param )* ;
    param: '?'? ID '*'? ;

    local: 'local' '?' ID ( ',' '?' ID )*;
    generator: 'generator' ID '*' ( ',' ID '*' )*;
    action: 'action' ('query' | 'remove' | 'add' ) argAction
                actionPreAdaptive*
                actionPostAdaptive* ;
        actionPreAdaptive: 'preAdaptiveFunction' ID '(' argLst ')' ;
        actionPostAdaptive: 'postAdaptiveFunction' ID '(' argLst ')' ;
        argAction: arg arg arg ;
        arg: '?'? ID '*'? ;
```

## Processamento de Funções Adaptativas

```
<<AutomatonBuilder::processamento de funções adaptativas>>=
@Override
public void enterActionPreAdaptive(AASParser.ActionPreAdaptiveContext ctx) {
    argLst = new ArrayList<>();
}

@Override
public void exitActionPreAdaptive(AASParser.ActionPreAdaptiveContext ctx) {
    argLst = null;
}

@Override
public void enterActionPostAdaptive(AASParser.ActionPostAdaptiveContext ctx) {
    argLst = new ArrayList<>();
}

@Override
public void exitActionPostAdaptive(AASParser.ActionPostAdaptiveContext ctx) {
```

```
    argLst = null;
}

@Override
public void enterArgAction(AASParser.ArgActionContext ctx) {
    argLst = new ArrayList<>();
}

@Override
public void exitArgAction(AASParser.ArgActionContext ctx) {
    argLst = null;
}
```