Melhorias na Sintaxe da Linguagem Jason

Jan Pierry Coelho dos Santos [1], **Jomi Fred Hübner**[2], **Jerusa Marchi**[1]

[1]Departamento de Informática e Estatística

[2]Departamento de Automação e Sistemas
Universidade Federal de Santa Catarina

janpierrycoelho@gmail.com, jomi.hubner@ufsc.br, jerusa.marchi@ufsc.br

Abstract. *Agent programming has been driven by several types of tools, such as the Jason programming language, which originally had academic purposes, but recently has also seen more use in real applications. However, throughout Jason's evolutionary process, several specific changes were made to it's grammar which, when reviewed in general, present some problems and potential for improvement. This article presents the problems identified and the improvements made in the grammar with a focus on simplicity, effectiveness and specificity without changing, as far as possible, the expressiveness of the language.*

Resumo. *A programação de agentes tem sido impulsionada por diversos tipos de ferramentas, como é o caso da linguagem de programação do Jason, que originalmente tinha fins acadêmicos, mas que vem sendo também utilizada em aplicações reais. No entanto, ao longo do processo de evolução do Jason, várias alterações pontuais foram feitas em sua gramática que, quando revistas de forma geral, apresentam alguns problemas e potencial de melhoria. Esse artigo apresenta alguns dos problemas identificados e as melhorias feitas na gramática com o foco em simplicidade, eficácia e especificidade sem alterar, na medida do possível, a expressividade da linguagem.*

1. Introdução

Assim como em outras áreas, a programação de agentes e sistemas multiagentes foi impulsionada pelo desenvolvimento de linguagens de programação e *frameworks* para fins específicos, como é o caso do Jason. O Jason é um *framework* para o desenvolvimento e simulação de sistemas multiagentes [Hübner et al. 2004, Bordini et al. 2007] que inicialmente tinha objetivos mais acadêmicos, mas que, com o passar do tempo, recebeu diversos usuários, tornando-se bastante utilizado em aplicações reais. É importante ressaltar que tanto o *framework*, quanto o interpretador e a linguagem que lhe são inerentes são chamados de Jason. A linguagem é uma extensão da linguagem de programação abstrata *AgentSpeak(L)* [Rao 1996] e é o foco principal deste trabalho.

O Jason teve sua primeira versão lançada no ano de 2004 [Hübner et al. 2004] e com o decorrer do tempo foi recebendo diversas atualizações[1]. Atualmente, ele se encontra na versão 2.6, sendo esta a versão utilizada como base deste trabalho. Contudo, para atender a crescente demanda dos sistemas multiagentes cada vez maiores e mais complexos, diversas mudanças foram feitas na linguagem Jason, muitas destas de forma

[1]http://jason.sourceforge.net

rápida visando atender às necessidades de usuários específicos. Ao longo do tempo, tais modificações resultaram em alguns problemas na estrutura da gramática da linguagem e na necessidade de um processo criterioso de revisão.

Desta forma, este trabalho tem como objetivos analisar a estrutura sintática da linguagem Jason de forma a identificar seus problemas, elaborar soluções para os problemas identificados, integrar tais soluções no código do interpretador Jason e por fim, realizar testes visando validar as mudanças e verificar se elas não impactam negativamente no correto funcionamento dos códigos já desenvolvidos na linguagem. Para tanto, o trabalho está estruturado da seguinte forma. Na Seção 2 é apresentada a estrutura sintática da linguagem Jason, apontando exemplos dos principais problemas encontrados. Na Seção 3 são apresentadas as soluções encontradas para os problemas relatados. As Seções 4 e 5 mostram como a nova estrutura sintática foi integrada ao nível semântico do Jason e a sua resposta com relação aos testes realizados. Por fim, a Seção 6 apresenta algumas considerações finais acerca do trabalho desenvolvido.

2. Estrutura sintática

A estrutura sintática de uma linguagem de programação é descrita por uma Gramática Livre de Contexto $G = (N, T, P, S)$ onde N é o conjunto de não-terminais ou categorias sintáticas, T é o conjunto de terminais, ou seja itens léxicos encontrados no texto fonte, como nomes dos agentes, nomes dos predicados e palavras reservadas. P é o conjunto de regras de produção que são estruturadas na forma $\alpha ::= \beta$ que significa que a forma sentencial $\alpha \in N$ será sobrescrita por β, onde $\beta \in \{N \cup T\}^*$[2]. Por fim, S se refere ao símbolo inicial da gramática, tal que $S \in N$.

A gramática de uma linguagem é normalmente descrita utilizando o formalismo Backus-Naur, do inglês *Backus-Naur formalism* (BNF), que padroniza e estende as regras de produção, permitindo uso de operadores como $*$ (fecho de Kleene) e $+$ (fecho positivo de Kleene) facilitando a definição das estruturas sintáticas.

A Figura 1 apresenta a gramática da linguagem Jason, onde é possível observar a constituição das regras de produção em BNF, como na produção `agent`. Já na produção `directive`, observam-se exemplos de categorias sintáticas, como `pred`, e terminais como ⟨`TK_BEGIN`⟩. É importante ressaltar que o item léxico `begin` encontrado no programa fonte será convertido no *token* `TK_BEGIN` quando analisado.

Após a análise e compreensão geral da estrutrura sintática da linguagem, tornou-se possível investigar as suas produções de forma a identificar os principais problemas.

2.1. Identificando Problemas

O interpretador Jason foi gerado utilizando o JavaCC[3], que é uma ferramenta *open source* para a geração de analisadores sintáticos ou *parsers* em código Java. *Parsers* gerados pelo JavaCC, não fazem uso de *backtracking* e, quando diante de um conflito de escolha entre duas ou mais opções de caminho de expansão, sempre escolherão a primeira. Para possibilitar a escolha diante de duas ou mais opções de caminhos de expansão possíveis,

[2]Onde $*$ é o fecho de Kleene e representa qualquer número de repetições de não-terminais e terminais concatenados, inclusive zero.

[3]https://javacc.github.io/javacc/

```
            agent ::= ( directive )* ( ( belief | initial_goal | plan )+ ( directive )* )* <EOF>
        directive ::= "{" ( <TK_BEGIN> pred "}" agent | pred "}" )
           belief ::= literal ( ":-" log_expr )? "."
     initial_goal ::= "!" literal "."
             plan ::= ( <TK_LABEL_AT> ( literal | list ) )? trigger ( ":" log_expr )? ( "<-" plan_body )? "."
          trigger ::= ( "+" | "-" | "^" ) ( ( "!" | "?" ) )? literal
        plan_body ::= plan_body_term ( ";" )? ( plan_body )?
   plan_body_term ::= plan_body_factor ( <TK_POR> plan_body_term )?
  plan_body_factor ::= ( stmtIF | stmtFOR | stmtWHILE | body_formula ) ( <TK_PAND> plan_body_factor )?
          stmtIF ::= <TK_IF> stmtIFCommon
     stmtIFCommon ::= "(" log_expr ")" rule_plan_term ( ( <TK_ELIF> stmtIFCommon | <TK_ELSE> rule_plan_term ) )?
         stmtFOR ::= <TK_FOR> "(" log_expr ")" rule_plan_term
        stmtWHILE ::= <TK_WHILE> "(" log_expr ")" rule_plan_term
     body_formula ::= ( "!" | "!!" | "?" | ( "+" ( ( "+" | "<" | ">" ) )? ) | ( "-" ( "+" | "-" )? ) )? ( log_expr )
   rule_plan_term ::= "{" ( ( <TK_LABEL_AT> ( pred | var ) )? trigger ( ":" log_expr )? ( ( "<-" | ";" ) )? )?
                          ( literal ":-" log_expr )? ( plan_body )? "}"
          literal ::= ( ( ( ( <ATOM> | var ) )? "::" )? ( <TK_NEG> )? ( pred | var ) ) | <TK_TRUE> | <TK_FALSE> )
             pred ::= ( <ATOM> | <TK_BEGIN> | <TK_END> ) ( "(" terms ")" )? ( list )?
            terms ::= term ( "," term )*
             term ::= log_expr
             list ::= "[" ( term_in_list ( "," term_in_list )* )? ( "|" ( <VAR> | <UNNAMEDVAR> | list ) )? "]"
     term_in_list ::= ( list | arithm_expr | string | rule_plan_term )
         log_expr ::= log_expr_trm ( "|" log_expr )?
     log_expr_trm ::= log_expr_factor ( "&" log_expr_trm )?
   log_expr_factor ::= ( <TK_NOT> log_expr_factor | rel_expr )
         rel_expr ::= ( arithm_expr | string | list | rule_plan_term )
                      ( ( "<" | "<=" | ">" | ">=" | "==" | "\\==" | "=" | "=.." )
                      ( arithm_expr | string | list | rule_plan_term ) )?
      arithm_expr ::= arithm_expr_trm ( ( "+" | "-" ) arithm_expr_trm )*
   arithm_expr_trm ::= arithm_expr_factor ( ( "*" | "/" | <TK_INTDIV> | <TK_INTMOD> ) arithm_expr_factor )*
arithm_expr_factor ::= arithm_expr_simple ( ( "**" ) arithm_expr_factor )?
arithm_expr_simple ::= ( <NUMBER> | "-" arithm_expr_simple | "+" arithm_expr_simple | "(" log_expr ")" | function )
          function ::= literal
               var ::= ( <VAR> | <UNNAMEDVARID> | <UNNAMEDVAR> ) ( list )?
            string ::= <STRING>
```

Figura 1. Gramática do Jason antes das alterações

o JavaCC permite que se defina um *lookahead* maior, ou seja, que o *parser* explore *tokens* mais à frente no fluxo de entrada, auxiliando a tomada de decisão [Aho et al. 2008].

Desta forma, para identificar problemas no interpretador Jason passou-se a observar os pontos de conflito existentes nas especificações léxica e sintática da linguagem Jason, que estão descritas no arquivo `AS2JavaParser.jj`. Este é o arquivo utilizado pelo JavaCC para produzir o *parser*. Foram identificados 12 pontos de conflito, relacionados a 4 *lookaheads* no código, cujos trechos de código são mostrados na Figura 2, 2 deles mais críticos, que verificam 47 e 150 *tokens* à frente do fluxo de entrada, e outros 2 mais simples que verificam apenas 4 *tokens* à frente.

O passo seguinte foi a análise de cada conflito. Dos 12 conflitos existentes, 8 estavam relacionados aos 2 *lookaheads* mais críticos. Os outros 4 conflitos eram relacionados a estrututura das produções da gramática, pois seus *lookaheads* não eram grandes.

Para auxiliar nesta análise, foram aplicados os conceitos de *First* e *Follow* [Aho et al. 2008]. O conjunto *First* identifica quais terminais podem iniciar uma forma sentencial a partir de um determinado não terminal que se encontra em análise, ou seja, qual é a produção que deve ser aplicada na derivação, logo é possível afirmar que existe um conflito de escolha se algum *token* do *First* de um dos caminhos de expansão também faz parte do *First* de outro caminho de expansão.

```
void directive() :
{}
{
  "{" ( LOOKAHEAD(4) <TK_BEGIN> pred() "}" agent() | pred() "}" )
}

void rule_plan_term():
{}
{
  "{"
    [ LOOKAHEAD(4) [ <TK_LABEL_AT> ( pred() | var() ) ] trigger() [ ":" log_expr() ] [ ( "<-" | ";" ) ] ]
    [ LOOKAHEAD(150) literal() ":-" log_expr() ]
    [ plan_body() ]
  "}"
}

void literal() :
{}
{
  ( [ LOOKAHEAD(47) [ ( <ATOM> | var() ) ] "::" ] [ <TK_NEG> ] ( pred() | var() ) ) | <TK_TRUE> | <TK_FALSE>
}
```

Figura 2. Lookaheads presentes na Gramática do Jason

Já o conjunto *Follow* permite saber, para cada não terminal, quais são os terminais que podem o suceder durante a avaliação, ou seja, permite identificar se um não terminal pode ser retirado da pilha de análise ou ainda quando a análise da forma sentencial derivada daquele não terminal acabou. Também é possível usar este conjunto para auxiliar na identificação do *First* em cenários específicos.

```
void agent() :
{}
{
  ( directive() )* ( ( belief() | initial_goal() | plan() )+ ( directive() )* )* <EOF>
}
```

Figura 3. Produção `agent` original.

Como exemplo do processo de análise dos conflitos, considere a produção `agent` apresentada na Figura 3, que é também o símbolo inicial da gramática da linguagem Jason e onde o ponto de conflito indicado pelo JavaCC se refere à parte onde ocorre a escolha entre os não-terminais `belief`, `initial_goal` e `plan`, cujos conjuntos *First* são os seguintes:

$$
\begin{aligned}
\mathit{First}(\texttt{belief}) &= \{\langle \texttt{ATOM}\rangle, \langle \texttt{VAR}\rangle, \langle \texttt{UNNAMEDVARID}\rangle, \langle \texttt{UNNAMEDVAR}\rangle, \\
&\quad \langle \texttt{TK\_NEG}\rangle, \langle \texttt{TK\_BEGIN}\rangle, \langle \texttt{TK\_END}\rangle, \langle \texttt{TK\_TRUE}\rangle, \\
&\quad \langle \texttt{TK\_FALSE}\rangle\} \\
\mathit{First}(\texttt{initial\_goal}) &= \{\text{“ ! ”}\} \\
\mathit{First}(\texttt{plan}) &= \{\langle \texttt{TK\_LABEL\_AT } \rangle, \text{“ + ”}, \text{“ - ”}, \text{“ ^ ”}\}
\end{aligned}
$$

Como é possível perceber na Figura 3 a escolha entre `belief`, `initial_goal` e `plan` possui o fechamento positivo de Kleene (+), indicando que o que está entre parênteses deve aparecer uma vez ou mais, por conta disso, após uma das opções aparecer, o *parser* tem como opção expandir novamente alguma delas ou seguir adiante e expandir `directive`. Levando isso em conta, também foi preciso considerar o *First* de `directive` para verificar o conflito.

$$\mathit{First}(\texttt{directive}) = \{\text{“ \{ ”}\}$$

Portanto, como não há *tokens* em comum entre os *Firsts* desses não-terminais, é possível afirmar que não existe conflito entre `belief`, `initial_goal`, `plan` e `directive`. Contudo, ainda assim é apresentado o aviso de conflito de escolha. A razão disso se dá pela forma como a produção `agent` foi estruturada, isso porque a parte `((belief | initial_goal | plan)+ (directive)*)*` possui o fechamento de Kleene (*) em volta, indicando que tudo entre parênteses pode aparecer quantas vezes forem necessárias, inclusive nenhuma, fazendo com que, caso apareça uma das produções das opções (`belief`, `initial_goal` e `plan`) no fluxo de entrada e na sequência apareça novamente mais uma delas, o *parser* não vai saber se deve expandir essa segunda a partir de `( belief | initial_goal | plan )+` ou se deve finalizar essa parte, desconsiderar `directive` e partir para mais uma iteração disso tudo. Por conta disso, é possível afirmar que colocar o fechamento positivo de Kleene em volta da escolha de `belief`, `initial_goal` e `plan` é redundante. Por fim, independentemente da escolha que o *parser* tome mediante este conflito, nenhuma delas resultará em um erro de interpretação, portanto o *lookahead* não é necessário, mas essa produção pode ser melhor estruturada, visando eliminar o conflito.

Além deste exemplo de conflito, um outro que vale destacar é o da produção `rule_plan_term`, apresentada na Figura 4, isso porque este conflito possui o maior *lookahead* encontrado na gramática do Jason, verificando 150 *tokens* à frente no fluxo de entrada. A causa do conflito se dá pela escolha que o *parser* tem entre expandir o não-terminal `literal` (na Figura 4, logo após o *LOOKAHEAD* (150), e expandir o não-terminal `plan_body` da linha seguinte, que também pode expandir `literal`. Dessa forma, o número de *tokens* a serem verificados à frente no fluxo de entrada deve levar em conta a quantidade de *tokens* que podem ser expandidos a partir de `literal`. No entanto, o que faz com que esse problema seja tão grave é que `literal` pode expandir o não-terminal `pred`, que pode expandir o não-terminal `list`, e que por fim pode expandir recursivamente mais não-terminais `list`. Por conta disso, tem-se que esse é um problema estrutural e que qualquer valor que venha a ser colocado no *lookahead* não resolverá este conflito de forma definitiva.

```
void rule_plan_term() :
{}
{
  "{"
    [ LOOKAHEAD(4)
      [ <TK_LABEL_AT> ( pred() | var() ) ]
      trigger()
      [ ":" log_expr() ] [ ( "<-" | ";" ) ]
    ]
    [ LOOKAHEAD(150) literal() ":-" log_expr() ]
    [ plan_body() ]
  "}"
}
```

Figura 4. Produção `rule_plan_term` original.

Para além da análise dos conflitos, para a identificação dos problemas considerou-se ainda a questão da especificidade das produções, ou seja, o quão bem uma produção representa o recurso da linguagem ao qual se propõe. Um bom exemplo disso se encontra na produção `directive`, cuja representação simplificada se encontra na Figura 5. De

forma a entender o problema em sua estrutura, é importante inicialmente ressaltar que as diretivas na linguagem Jason podem se dar de duas formas: com `begin` e `end`, e sem `begin` e `end`. Logo, a partir do ponto que uma diretiva inicia com *begin*, ela obrigatoriamente deve finalizar com *end*. Contudo, como é possível perceber na Figura 5, a produção `directive` possui um *token* ⟨`TK_BEGIN`⟩ mas não possui um *token* ⟨`TK_END`⟩. A razão para que mesmo assim ela ainda funcione é que, após o não-terminal `pred` e o *token* "}", há o não-terminal `agent`, que tem a opção de expandir novamente o não-terminal `directive`, que pode seguir pelo não-terminal `pred` ao final da produção, que pode então expandir o *token* ⟨`TK_END`⟩. No entanto, o *parser* pode expandir outros *tokens* no lugar de ⟨`TK_END`⟩ em `pred`, como ⟨`ATOM`⟩ ou até mesmo outro ⟨`TK_BEGIN`⟩. Contudo, como foi comentado, é imprescindível que uma diretiva que inicie com `begin` termine com `end`, por conta disso, já que o *parser* considera como válida uma diretiva que não atende a esse requisito é possível afirmar que essa produção não é específica.

```
void directive() :
{}
{
  "{" ( LOOKAHEAD(4) <TK_BEGIN> pred() "}" agent() | pred() "}" )
}
```

Figura 5. Produção `directive` original.

A existência desse tipo de inconformidade não implica necessariamente que códigos mais abrangentes, que não se adequam ao que o recurso da linguagem realmente representa, vão ser considerados válidos, pois ainda é possível que a análise semântica venha a detectar inconformidades como essa. No entanto, é preciso considerar que estes são problemas estruturais e portanto deveriam ser tratados pelo analisador sintático, pois cabe ao analisador semântico tratar apenas de questões relacionadas ao significado dos comandos, como por exemplo a incompatibilidade de tipos ou redeclarações de variáveis.

3. Soluções Elaboradas

A seção anterior ilustrou três exemplos de problemas encontrados após a análise da gramática do Jason. O passo seguinte foi elaborar soluções buscando eliminar os conflitos e melhorar a estrutura das produções. Para tanto, três características principais foram consideradas na construção das soluções: simplicidade, eficácia e especificidade.

3.1. Simplicidade

Para que a estrutura da gramática se tornasse mais organizada e legível, as melhorias foram pensadas de forma que as produções ficassem as mais simples possíveis. Como exemplo, reconsidere a produção `agent` apresentada na Figura 3. Como comentado anteriormente, a sua estrutura original é mais complexa do que o necessário, já que o fechamento positivo de Kleene englobando a escolha entre `befief`, `initial_goal` e `plan` é redundante. Todavia, por mais que a sua remoção resolva o conflito, também é possível reestruturar a produção de forma que se obtenha uma produção mais simples, como apresentado na Figura 6. Segundo esta nova estrutura, um agente pode ser definido como uma sequência de `directive`, `belief`, `initial_goal` e `plan` podendo aparecer quantas vezes forem necessárias e em qualquer ordem. A nova produção não é nem mais restritiva e nem abrangente que a especificação original e por conta disso é possível afirmar que ambas possuem o mesmo valor sintático.

```
void agent() :
{}
{
  ( directive() | belief() | initial_goal() | plan() )* <EOF>
}
```

Figura 6. Produção `agent` modificada.

3.2. Eficácia

Além da simplicidade, outro aspecto levado em consideração foi a eficácia das produções. Isso porque, como ilustrado no caso da produção `rule_plan_term` (apresentada na Figura 4), existiam conflitos na gramática que, mesmo com o aumento do *lookahead*, não seriam resolvidos. Para exemplificar como se deu a solução desse tipo de problema, considere a produção alterada apresentada na Figura 7. A nova estrutura possui outras alterações focadas em resolver outros problemas na produção, contudo, a mudança que foi feita para resolver o problema da eficácia se encontra na parte `plan_body` [":-" `log_expr`]. Esta parte foi modificada de forma a representar ambas as opções conflitantes deste contexto, que eram os recursos da linguagem: `rule_term`, representado por `literal` ":-" `log_expr`; e `plan_body_only` que, como o próprio nome diz, consiste apenas do não-terminal `plan_body`. O recurso `plan_body_only` pode ser representado expandindo `plan_body` e desconsiderando a parte opcional [":-" `log_expr`], e o recurso `rule_term` pode ser representado com o `literal` sendo expandido através de `plan_body`, que era a causa do conflito apresentado na Figura 4, e depois expandindo a parte opcional [":-" `log_expr`].

```
void rule_plan_term() :
{}
{
    "{"
      [
        [ plan_term_annotation() ]
            trigger() [ ":" log_expr() ] [ ( ";" [ plan_body() ] | "<-" plan_body() ) ]
        | plan_body() [ ":-" log_expr() ]
      ]
    "}"
}
```

Figura 7. Produção `rule_plan_term` modificada.

No entanto essa nova estrutura é uma representação mais abrangente do que a anterior, já que permite que outros *tokens* que possam ser expandidos através de `plan_body` sejam seguidos de ":-" `log_expr`, o que na prática não deve ser permitido. Esse problema foi solucionado com uma anotação de código que verifica se, quando a parte que é exclusiva de `rule_term` está presente, `plan_body` é composto de apenas um único `literal`, reportando um erro nesse caso. Por mais que a solução de problemas sintáticos através de anotações de código não seja ideal, o caso de `rule_plan_term` era bastante complexo e exigiu tal abordagem visando sua solução definitiva.

3.3. Especificidade

O último pilar teve como objetivo que as produções fossem representações mais fiéis dos seus respectivos recursos da linguagem. Exemplos de melhorias elaboradas com esse

foco podem ser observadas no caso da produção `directive`, apresentada na Figura 5. Como comentado anteriormente, esta produção possui problemas de especificidade por não restringir sintaticamente que diretivas que iniciam com *begin* terminem com *end*. A produção foi reestruturada conforme apresentado na Figura 8.

```
void directive() :
{}
{
    "{" (
          <TK_BEGIN> directive_argument() "}" ( agent_component() )* "{" <TK_END> "}"
          | directive_argument() "}"
        )
}
```

Figura 8. Produção `directive` modificada.

Assim como em `rule_plan_term`, essa produção possui mais alterações focadas em resolver outros problemas. Contudo, a solução para o problema de especificidade pode ser observada no primeiro caminho de expansão dessa produção, onde têm-se que ao iniciar a diretiva com o *token* ⟨`TK_BEGIN`⟩ é obrigatório que exista o fim dessa diretiva com o *token* ⟨`TK_END`⟩.

3.4. Resultado da nova estrutura

Ao juntar essas e as outras produções que tiveram correções com as produções que não foram alteradas, obteve-se a nova estrutura sintática da linguagem Jason. Ao gerar o *parser* foi constatada a existência de 6 novos conflitos de escolha para os quais, novamente, foi necessária a análise individual de forma a verificar se não representam problemas para o correto funcionamento do interpretador.

Destes novos conflitos, 2 puderam ser solucionados com o uso de lookaheads que verificam apenas 2 *tokens* à frente no fluxo de entrada, outros 3 já existiam na estrutura original do interpretador Jason e não necessitaram de tratamento pois o primeiro caminho de expansão possível sempre é o correto por uma questão de precedência de operadores, e por fim, o último conflito se referia a um problema de eficácia da estrutura original e foi solucionado através de um recurso do JavaCC que permite o uso de não-terminais no *lookahead* no lugar de um número de *tokens*, como mostra a Figura 9.

```
void literal() :
{}
{
    [ LOOKAHEAD(namespace()) namespace() ] [ <TK_NEG> ] ( pred() | var() ) | <TK_TRUE> | <TK_FALSE>
}
```

Figura 9. Produção `literal` modificada.

Essa modificação faz com que o *parser* só expanda o não-terminal `namespace` quando tiver certeza de que o código que está sendo analisado realmente representa este não-terminal. Desta forma, concluiram-se as alterações possíveis visando melhor estruturar a gramática da linguagem Jason. A nova gramática é apresentada na Figura 10.

4. Integração

A última etapa foi a integração na nova estrutura sintática ao interpretador Jason, gerando uma nova versão da linguagem. Para isto foi necessário analisar a ligação que existia

```
              agent ::= ( agent_component )* <EOF>
    agent_component ::= ( directive | belief | initial_goal | plan )
          directive ::= "{" ( <TK_BEGIN> directive_arguments "}" ( agent_component )* "{" <TK_END> "}" | directive_arguments "}" )
 directive_arguments ::= <ATOM> ( "(" terms ")" )? ( list )?
             belief ::= literal ( ":-" log_expr )? "."
       initial_goal ::= "!" literal "."
               plan ::= ( plan_annotation )? trigger ( ":" log_expr )? ( "<-" plan_body )? "."
    plan_annotation ::= <TK_LABEL_AT> ( ( <ATOM> | <TK_BEGIN> | <TK_END> ) ( list )? | list )
            trigger ::= ( "+" | "-" | "^" ) ( ( "!" | "?" ) )? literal
          plan_body ::= ( plan_body_term ( ";" ( plan_body )? )? | statement ( ";" )? ( plan_body )?
     plan_body_term ::= plan_body_factor ( <TK_POR> plan_body_term )?
   plan_body_factor ::= body_formula ( <TK_PAND> plan_body_factor )?
          statement ::= ( stmtIF | stmtFOR | stmtWHILE )
             stmtIF ::= <TK_IF> stmtIFCommon
       stmtIFCommon ::= "(" log_expr ")" "{" ( stmt_body )? "}" ( <TK_ELIF> stmtIFCommon | <TK_ELSE> "{" ( stmt_body )? "}" )?
            stmtFOR ::= <TK_FOR> "(" log_expr ")" "{" ( stmt_body )? "}"
          stmtWHILE ::= <TK_WHILE> "(" log_expr ")" "{" ( stmt_body )? "}"
          stmt_body ::= plan_body
       body_formula ::= ( ( ( "!" | "!!" ) literal ) | ( ( "?" | "+" ( "+" | "<" | ">" )? | "-" ( "+" | "-" )? )? log_expr ) )
     rule_plan_term ::= "{" ( ( plan_term_annotation )? trigger ( ":" log_expr )? ( ( ";" ( plan_body )? | "<-" plan_body ) )?
                          | plan_body ( ":-" log_expr )? )? "}"
plan_term_annotation ::= <TK_LABEL_AT> ( ( <ATOM> | <TK_BEGIN> | <TK_END> ) ( list )? | var | list )
            literal ::= ( ( namespace )? ( <TK_NEG> )? ( pred | var ) | <TK_TRUE> | <TK_FALSE> )
          namespace ::= ( <ATOM> | var )? "::"
               pred ::= ( <ATOM> | <TK_BEGIN> | <TK_END> ) ( "(" terms ")" )? ( list )?
              terms ::= term ( "," term )*
               term ::= log_expr
               list ::= "[" ( term_in_list ( "," term_in_list )* )? ( "|" ( <VAR> | <UNNAMEDVAR> | list ) )? "]"
       term_in_list ::= ( list | arithm_expr | string | rule_plan_term )
           log_expr ::= log_expr_trm ( "|" log_expr )?
       log_expr_trm ::= log_expr_factor ( "&" log_expr_trm )?
    log_expr_factor ::= ( <TK_NOT> )? rel_expr
           rel_expr ::= ( arithm_expr | string | list | rule_plan_term ) ( ( "<" | "<=" | ">" | ">=" | "==" | "\\==" | "=" | "=.." )
                        ( arithm_expr | string | list | rule_plan_term ) )?
        arithm_expr ::= arithm_expr_trm ( ( "+" | "-" ) arithm_expr_trm )*
    arithm_expr_trm ::= arithm_expr_factor ( ( "*" | "/" | <TK_INTDIV> | <TK_INTMOD> ) arithm_expr_factor )*
 arithm_expr_factor ::= arithm_expr_simple ( ( "**" ) arithm_expr_factor )?
 arithm_expr_simple ::= ( "+" | "-" )? ( <NUMBER> | ( "(" log_expr ")" ) | function )
           function ::= literal
                var ::= ( <VAR> | <UNNAMEDVARID> | <UNNAMEDVAR> ) ( list )?
             string ::= <STRING>
```

Figura 10. Nova gramática do Jason.

entre a estrutura original da linguagem e as anotações de código que fazem a verificação semântica e a geração de código. A dificuldade de integração das produções varia dependendo da complexidade e também do tamanho das mudanças realizadas. Um exemplo simples que ilustra bem como se deu o processo de integração é o caso da produção `stmtFOR`, apresentada na Figura 11.

Como resultado das melhorias feitas na estrutura desta produção, ao invés de utilizar `rule_plan_term` como corpo do comando, é utilizado "{" `stmt_body` "}", como mostra a Figura 12. Essa mudança foi feita pois a linguagem Jason só permite que os comandos `if`, `for` e `while` tenham como corpo o não-terminal `plan_body` e `rule_plan_term` permitia que outras coisas fossem expandidas. Desta forma, o código que era necessário para verificar se `rule_plan_term` era composto apenas de `plan_body` pôde ser removido e em seu lugar foi adicionada uma verificação para o cenário do corpo estar vazio em `stmtFOR`.

De forma similar, procedeu-se às demais alterações visando a integração entre as partes sintática e semântica.

```
PlanBody stmtFOR() :   { Object B; Term T1; Literal stmtLiteral; }
{
  <TK_FOR>
  "("
  B = log_expr()
  ")"
  T1 = rule_plan_term()

            { try {
                if (T1.isRule()) {
                  throw new ParseException(getSourceRef(T1)+"for requires a plan body.");
                }
                stmtLiteral =
                  new InternalActionLiteral(ASSyntax.createStructure(".foreach", (Term)B, T1), curAg);
                stmtLiteral.setSrcInfo( ((Term)B).getSrcInfo() );
                return new PlanBodyImpl(BodyType.internalAction, stmtLiteral);
              } catch (Exception e) {
                e.printStackTrace();
              }
            }
}
```

Figura 11. Integração da produção `stmtFOR` original.

```
PlanBody stmtFOR() :
{
  Object B; PlanBody pb = null; Literal stmtLiteral;
}
{
  <TK_FOR> "(" B = log_expr() ")" "{" [ pb = stmt_body() ] "}"

            {
              try {
                if (pb == null) {
                  pb = new PlanBodyImpl();
                }
                stmtLiteral =
                  new InternalActionLiteral(ASSyntax.createStructure(".foreach", (Term)B, pb), curAg);
                stmtLiteral.setSrcInfo( ((Term)B).getSrcInfo() );
                return new PlanBodyImpl(BodyType.internalAction, stmtLiteral);
              } catch (Exception e) {
                e.printStackTrace();
              }
            }
}
```

Figura 12. Integração da produção `stmtFOR` modificada.

5. Testes

Foram diversas as mudanças implementadas, resultando em um novo arquivo do *parser* bastante distinto do original. Para assegurar que a nova versão do Jason suporte os códigos já desenvolvidos, não afetando os seus usuários, foram feitos testes de funcionamento. O Jason possui uma série de testes que foram implementados com o objetivo de validar seu funcionamento após mudanças em sua estrutura. Esses testes foram utilizados neste trabalho. O processo de integração e testes ocorreu de forma concomitante e, em caso de falha nos testes, ajustes na gramática e nas ações semânticas foram realizados. Eventualmente, alguma modificação foi necessária nos testes, pois ao tornar algumas produções mais específicas, algumas construções deixaram de ser válidas. Abaixo são listadas algumas construções que agora são inválidas na linguagem e a forma nova, que é aceita pelo *parser*.

5.1. Caso do *token* $\langle TK\_NOT \rangle$

Na definição original, era possível usar o `not` diversas vezes em sequência, como em `not not expressão`, devido a estrutura da produção `log_expr_factor` (veja Figura 1).

Com a alteração desta produção conforme apresentado na Figura 10, o comando passa a ser `not (not expressão)`.

5.2. Caso dos Operadores "+" e "-"

De forma similiar ao caso anterior, a definição original permite que se declare sequências de "+" e "-" antes de seguir para um número, função ou literal devido a estrutura da produção `arithm_expr_simple`, conforme apresentado na Figura 1. Um exemplo possível desta construção é `+ + - - - + number`. Com a nova estrutura desta produção, apresentada na Figura 10, tal construção não é mais possível.

5.3. Caso da construção `plan_body`

A terceira construção, ao contrário dos anteriores, tem um impacto significativo, pois modifica a forma como o corpo dos planos é declarado. Na gramática original, a produção `plan_body` (veja Figura 1) permite que os termos do corpo sejam separados ou não por "`;`" e também permitia o "`;`" depois do último comando, antes do "`.`", conforme ilustra a Figura 14.

```
+!plano
   <-
   !primeiroComando
   ?segundoComando;
   !terceiroComando;.
```

Figura 13. Exemplo de plano usando a produção `plan_body` original.

Contudo, para evitar problemas de conflitos optou-se por tornar o "`;`" obrigatório entre os termos do corpo, com exceção dos comandos `if | for | while`. Além disso, o "`;`" aparece apenas separando os termos, logo o último termo não possui o "`;`" apenas o "`.`". A nova versão da produção `plan_body` pode ser vista na Figura 10. Abaixo segue um exemplo utilizando a nova estrutura:

```
+!plano
   <-
   !primeiroComando;
   ?segundoComando;
   !terceiroComando.
```

Figura 14. Exemplo de plano usando a produção `plan_body` modificada.

6. Considerações finais

Este trabalho teve como objetivo inicial a análise e a refatoração do *parser* da linguagem Jason. Para realizar a análise, foi necessária uma ampla compreensão acerca das estruturas da linguagem e a identificação dos problemas teve como ponto de partida os conflitos existentes na gramática original. Dos 12 conflitos identificados e investigados, os 9 principais foram mitigados considerando três pilares: simplicidade, eficácia e especificidade. Outros 3 não foram tratados pois o primeiro caminho de expansão possível da produção é

sempre o correto. Numa segunda etapa, foram identificados 6 conflitos, 3 dos quais foram tratatos e os outros 3 são os mesmos citados anteriormente. Além dos conflitos, também foram reestruturadas diversas produções, visando atender aos critérios supracitados.

A nova gramática elaborada, foi validada através de testes disponíveis no próprio Jason de forma a garantir que o resultado final fosse uma versão do Jason que suportasse grande parte dos códigos já desenvolvidos pela comunidade de usuários.

As mudanças realizadas na gramática, em sua grande maioria, não implicam mudanças no funcionamento externo do Jason. Contudo, algumas delas realmente limitam o que pode ser escrito em códigos Jason na prática. No entanto, essas mudanças foram pensadas de forma a tornar a linguagem mais específica e organizada.

De forma geral, é possível afirmar que as mudanças realizadas impactaram positivamente a linguagem, pois sua estrutura ficou mais organizada, legível e específica e, em alguns casos, mais eficaz que a estrutura original, pois algumas mudanças resolveram problemas de corretude existentes na estrutura original. Assim a nova gramática facilitará futuras manutenções e expansões, assegurando que o Jason possa continuar sendo uma importante linguagem no desenvolvimento de sistemas multiagentes.

A nova versão do Jason, contendo as alterações realizadas está disponível no repositório `http://jason.sourceforge.net`.

Referências

Aho, A., Lam, M., Sethi, R., and Ullman, J. (2008). *Compiladores - Princípios, técnicas e ferramentas*. Pearson Addison-Wesley, 2nd edition.

Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons.

Hübner, J. F., Bordini, R. H., and Vieira, R. (2004). Introdução ao desenvolvimento de sistemas multiagentes com jason. *XII Escola de Informática da SBC*, 2:51–89.

Rao, A. S. (1996). Agentspeak(l): Bdi agents speak out in a logical computable language. In Van de Velde, W. and Perram, J. W., editors, *Agents Breaking Away*, pages 42–55, Berlin, Heidelberg. Springer Berlin Heidelberg.