4
Mecanismo de Rastreamento Baseado em Transformações

Este capítulo apresenta a proposta do mecanismo de rastreamento centrado na utilização da tecnologia transformacional. Este mecanismo permite a captura das informações de rastreamento relacionadas às ligações Evolui-Para e Justifica, do modelo de referência proposto por Ramesh & Jarke [3], entre artefatos produzidos durante o processo de desenvolvimento de software.

A figura 27 mostra uma visão geral do mecanismo proposto. Ele é composto pelas atividades de Geração das Transformações de Rastreamento (*Generate Trace Transformation*) e de Refinamento de Modelos (*Refine*).

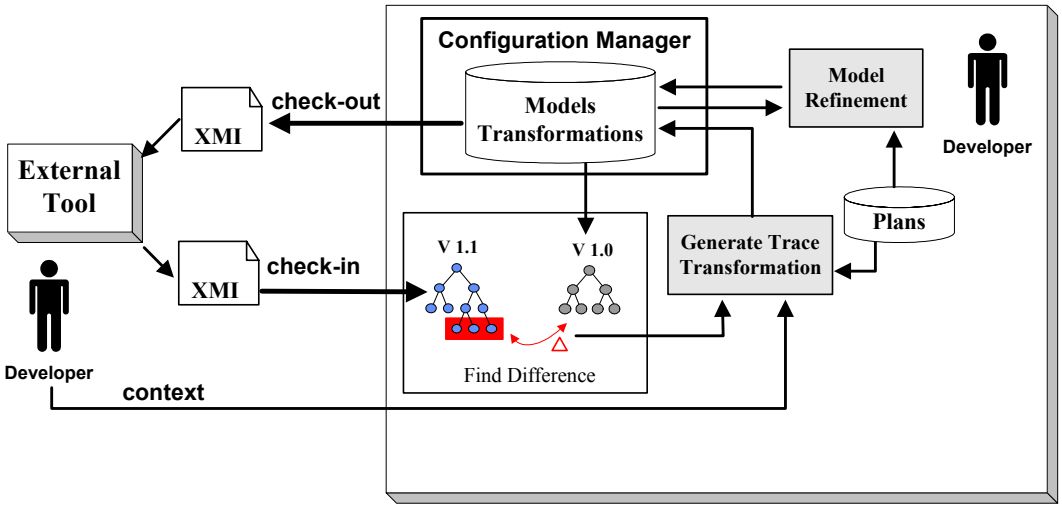

Figura 27 – Visão Geral do Mecanismo de Rastreamento

O uso do mecanismo se inicia a partir da obtenção (*check-out*) de um conjunto de artefatos armazenados e gerenciados pelo mecanismo de rastreamento. Estes artefatos são modificados pelos desenvolvedores, que realizam uma série de operações sobre eles utilizando alguma ferramenta externa. Os artefatos modificados são então submetidos ao mecanismo de rastreamento (*check-in*) onde é executada a atividade de geração das transformações de rastreamento. Esta atividade tem como objetivo gerar um conjunto de transformações que represente as mudanças realizadas pelos desenvolvedores ao evoluírem os artefatos de sua versão inicial, obtida no momento do *check-out*, para a versão subseqüente, inserida no momento do *check-in*. A geração destas

transformações baseia-se na comparação das duas versões do artefato, identificando quais mudanças foram realizadas pelo desenvolvedor.

Estas mudanças constituem o *delta* entre as versões e representam o resultado de um conjunto de ações realizadas pelo desenvolvedor. O mecanismo prevê ainda a utilização de uma técnica de reconhecimento de planos [67] procurando identificar, dentre os planos preexistentes, aquele que fornece um resultado similar ao expresso no *delta* que foi anteriormente computado. O plano identificado fornece informações sobre qual o objetivo do desenvolvedor ao realizar a mudança do artefato original e, em conjunto com informações do contexto do trabalho do desenvolvedor (por exemplo, sua função e a fase atual do processo de desenvolvimento), qual a justificativa para a ação realizada. O armazenamento destas informações e o controle de versões são realizados pelo Gerenciador de Configurações (CM). Se nenhum plano for identificado durante esta atividade um novo plano pode ser criado pelo desenvolvedor e utilizado em outras interações.

A atividade de refinamento do modelo permite a aplicação das transformações anteriormente geradas de maneira a evoluir automaticamente um artefato de uma versão para a versão seguinte. A aplicação destas transformações pode ser realizada nos próprios artefatos utilizados como base na sua geração, permitindo que o gerenciador de configuração armazene apenas uma versão inicial e as transformações que levem a nova versão, ou podem ainda ser aplicadas no próprio processo de desenvolvimento de um novo artefato. Neste caso o desenvolvedor especifica o contexto do seu trabalho, seleciona um plano dentre um conjunto de planos aplicáveis à situação e que foram identificados pelo sistema, e finalmente o sistema aplica as transformações ao artefato em desenvolvimento, conseguindo desta maneira a reutilização do plano e das transformações geradas em um novo contexto.

Na próxima seção será apresentada a representação escolhida para os artefatos e nas seções seguintes o detalhamento das atividades do processo de rastreamento. Na conclusão deste capítulo será feita uma análise do mecanismo proposto em relação aos requisitos desejáveis para sistemas de rastreamento apresentados na seção 3.3.2.

4.1. Representação dos Artefatos

A linguagem XML e tecnologias relacionadas são bastante utilizadas pela indústria, existindo diversas aplicações reais que as utilizam. Particularmente no desenvolvimento de software, o formato XMI* está sendo utilizado por diversas ferramentas de suporte ao desenvolvimento de software para expressar o conteúdo de modelos orientados a objetos (OO) em um formato que permita a troca de informações entre elas.

De maneira a aumentar o potencial de uso do mecanismo de rastreamento proposto, permitindo o uso de ferramentas externas na edição dos artefatos, optou-se por utilizar a linguagem XML na descrição dos artefatos do sistema. O tratamento destes artefatos descritos em XML pela Máquina Draco-PUC é viabilizado pela utilização do mecanismo de integração de domínios Draco-PUC / XML apresentado na seção 2.1.3, que cria de maneira automática um domínio Draco-PUC para cada definição de aplicação XML.

O mecanismo proposto pode ser utilizado no rastreamento de quaisquer tipos de artefatos descritos em XML, inclusive no formato XMI que é adotado pelas principais ferramentas de suporte ao desenvolvimento orientado a objetos, tais como o Rational Rose [68] e o Together [69].

A implementação do mecanismo de rastreamento realizada nesta tese foi direcionada para o modelo de cenários e do Léxico Ampliado da Linguagem (LAL) mostrados na seção 3.1. A definição do tipo do documento utilizado (DTD) na definição dos cenários é mostrada na figura 28 onde, além dos elementos anteriormente apresentados para um cenário, foi adicionado o elemento *reference* que permite que se estabeleçam ligações de correspondência entre os termos utilizados e suas definições no LAL. Este elemento é formado pelos atributos *name* (obrigatório) que indica qual o símbolo que está sendo referenciado, *address* (opcional) que indica algum ponto especifico dentro do artefato, *version* (obrigatório) que especifica a versão do artefato referenciado, e por último *label*

* XMI especifica um modelo aberto de troca de dados de modelos definido pelo Object Management Group (OMG). Seu objetivo é fornecer aos desenvolvedores que utilizam a tecnologia de orientação a objetos a habilidade de trocar dados pela Internet de uma forma padronizada.

(opcional) que é um sinônimo para o artefato referenciado utilizado no contexto do local onde a referência foi inserida.

```
<!ELEMENT scenario (title , goal* , context* , actor* , resource* , episode*)>
<!ELEMENT title (#PCDATA | reference)*>
<!ELEMENT goal (#PCDATA | reference)*>
<!ATTLIST goal  address CDATA  #IMPLIED >
<!ELEMENT context (#PCDATA | reference)*>
<!ATTLIST context  address CDATA  #IMPLIED >
<!ELEMENT actor (#PCDATA | reference)*>
<!ATTLIST actor  address CDATA  #IMPLIED >
<!ELEMENT resource (#PCDATA | reference)*>
<!ATTLIST resource  address CDATA  #IMPLIED >
<!ELEMENT episode ((definition | execute) , exception* , restriction*)>
<!ATTLIST episode  address CDATA  #IMPLIED >
<!ELEMENT definition (#PCDATA | reference)*>
<!ELEMENT execute (reference)>
<!ELEMENT exception (trigger , (definition | execute))>
<!ATTLIST exception  address CDATA  #IMPLIED >
<!ELEMENT trigger (#PCDATA | reference)*>
<!ELEMENT restriction (#PCDATA | definition | execute)*>
<!ATTLIST restriction  address CDATA  #IMPLIED >
<!ELEMENT reference EMPTY>
<!ATTLIST reference name CDATA #REQUIRED>
<!ATTLIST reference address CDATA #IMPLIED>
<!ATTLIST reference version CDATA #REQUIRED>
<!ATTLIST reference label CDATA #IMPLIED>
```

Figura 28 – Definição do tipo do documento (DTD) de descrição de cenários

Exemplificando o uso da DTD para cenários, a figura 29 mostra um cenário com seus elementos. Cada termo sublinhado faz referência a uma entrada do LAL. A descrição deste cenário em XML é apresentada na figura 30, onde podemos notar os elementos *reference* utilizados para referenciar a definição dos termos no LAL.

Scenario: malfunction occurs

Goal	how to proceed when a malfunction occurs
Context	4th floor of building 32
Actor	Control system
Actor	facility manager
Resource	Ceiling light groups
Resource	Control panel
Episode	inform facility manager of malfunction
Episode	inform user of malfunction
Episode	IF Hallway section is controlable neither automatically nor manually THEN Ceiling light groups have to be on
Episode	Find reason for malfunction
Episode	facility manager can correct manually malfunction undetected by the Control system

Figura 29 – Cenário *malfunction occurs*

```
<scenario>
  <title>malfunction occurs</title>
  <goal address="1">how to proceed when a<reference name="malfunction" version="1"/>occurs</goal>
  <context address="2"><reference name="4th floor of building 32" version="1"/></context>
  <actor address="3"><reference name="Control system" version="1"/></actor>
  <actor address="4"><reference name="facility manager" version="1"/></actor>
  <resource address="5"><reference name="Ceiling light groups" version="1"/></resource>
  <resource address="6"><reference name="Control panel" version="1"/></resource>
  <episode address="7">
    <definition>inform<reference name="facility manager" version="1"/>of<reference name="malfunction"
version="1"/></definition>
  </episode>
  <episode address="8">
    <definition>inform<reference name="user" version="1"/>of<reference name="malfunction" version="1"/>
</definition>
  </episode>
  <episode address="9">
    <definition>IF<reference name="Hallway section" version="1"/>is controlable neither automatically nor
manually THEN<reference name="Ceiling light groups" version="1"/>have to be on</definition>
  </episode>
  <episode address="10">
    <definition>Find reason for<reference name="malfunction" version="1"/></definition>
  </episode>
  <episode address="11">
    <definition><reference name="facility manager" version="1"/>can correct manually<reference
name="malfunction" version="1"/>undetected by the<reference name="Control system" version="1"/></
definition>
  </episode>
</scenario>
```

Figura 30 – Descrição XML do cenário *malfunction occurs*

```
scenario : ST 'scenario'  ET (title goal* context* actor* resource* episode*) '</scenario>' ;
title : ST 'title'  ET (text|reference)* '</title>' ;
goal : ST 'goal' att_address_goal ET (text|reference)* '</goal>' ;
context : ST 'context' att_address_context ET (text|reference)* '</context>' ;
actor : ST 'actor' att_address_actor ET (text|reference)* '</actor>' ;
resource : ST 'resource' att_address_resource ET (text|reference)* '</resource>';
episode : ST 'episode'  att_address_episode ET ((definition|execute) exception* restriction*) '</episode>' ;
definition : ST 'definition'  ET (text|reference)* '</definition>' ;
execute : ST 'execute'  ET (reference) '</execute>' ;
exception : ST 'exception'  att_address_exception ET (trigger (definition|execute)) '</exception>' ;
trigger : ST 'trigger'  ET (text|reference)* '</trigger>' ;
restriction : ST 'restriction' att_address_restriction ET (text|definition|execute)* '</restriction>' ;
reference : ST 'reference' ( att_name_reference | att_address_reference | att_version_reference |
att_label_reference )* EmptyEndTag ;
att_address_goal : 'address' '='  String   | ;
att_address_context : 'address' '='  String   | ;
att_address_actor : 'address' '='  String   | ;
att_address_resource : 'address' '='  String   | ;
att_address_episode : 'address' '='  String   | ;
att_address_exception : 'address' '='  String   | ;
att_address_restriction : 'address' '='  String   | ;
att_name_reference : 'name' '='  String  ;
att_address_reference : 'address' '='  String ;
att_version_reference : 'version' '='  String ;
att_label_reference : 'label' '='  String ;
text           : Text ;
ST       : "<"       { setInsideTAG(); } ;
ET         : ">"       { setOutsideTAG(); } ;
EmptyEndTag      : "/>"       { setOutsideTAG(); } ;
String          : \"([^\"])*\" ;
String          : \'([^\'])*\' ;
Text           : <REG_OUTSIDETAG>[^\>\<\{\}\[\]\?]*[^ \n\t\>\<\{\}\[\]\?]+[^\>\<\{\}\[\]\?]* ;
Text           : "??"[^\>\<\{\}\[\]\?]*"??" ;
```

Figura 31 – Definição da sintaxe do domínio de cenários na Máquina Draco-PUC

```
<html xmlns:java="http://xml.apache.org/xslt/java">
<head><title>Scenarios Descriptions</title></head>
<body>
 <table width="100%">
   <tr><td width="10%"><b><font size="+2" color="#ff0000">Scenario:</font></b></td>
       <td><b><font size="+2" color="#ff0000">malfunction occurs</font></b></td></tr>
   <tr><td><b>Goal</b></td>
     <td>how to proceed when a<a href="html/symbols/versions/10037.html">malfunction</a>occurs</td>
   </tr>
   <tr><td><b>Context</b></td>
     <td><a href="html/symbols/versions/10002.html">4th floor of building 32</a></td></tr>
   <tr><td><b>Actor</b></td>
     <td><a href="html/symbols/versions/10014.html">Control system</a></td></tr>
   <tr><td><b>Actor</b></td>
     <td><a href="html/symbols/versions/10027.html">facility manager</a></td></tr>
   <tr><td><b>Resource</b></td>
     <td><a href="html/symbols/versions/10009.html">Ceiling light groups</a></td></tr>
   <tr><td><b>Resource</b></td>
     <td><a href="html/symbols/versions/10013.html">Control panel</a></td></tr>
   <tr><td valign="Top"><b>Episode</b></td>
     <td valign="Top">inform <a href="html/symbols/versions/10027.html">facility manager</a>
 of <a href="html/symbols/versions/10037.html">malfunction</a></td></tr>
   <tr><td valign="Top"><b>Episode</b></td>
     <td valign="Top">inform <a href="html/symbols/versions/10063.html"> user</a>
of <a href="html/symbols/versions/10037.html">malfunction</a></td></tr>
   <tr><td valign="Top"><b>Episode</b></td>
     <td valign="Top">IF <a href="html/symbols/versions/10031.html">Hallway section</a>
is controlable neither automatically nor manually THEN <a href="html/symbols/versions/10009.html"> Ceiling
light groups</a>have to be on</td></tr>
   <tr><td valign="Top"><b>Episode</b></td>
     <td>Find reason for<a href="html/symbols/versions/10037.html">malfunction</a></td></tr>
   <tr><td valign="Top"><b>Episode</b></td>
     <td valign="Top"><a href="html/symbols/versions/10027.html">facility manager</a> can correct
manually <a href="html/symbols/versions/10037.html"> malfunction</a> undetected by
the <a href="html/symbols/versions/10014.html"> Control system</a></td></tr>
  </table></body></html>
```

Figura 32 – Representação HTML para o cenário *malfunction occurs*

De maneira a permitir a manipulação de cenários descritos em XML pela Máquina Draco-PUC foi aplicado o mecanismo de importação de domínios, apresentado na seção 2.1.3, criando-se um domínio de cenários na Máquina Draco-PUC correspondente à estrutura definida na DTD. A descrição da sintaxe deste domínio é mostrada na figura 31 utilizando-se o domínio de definição da gramática (GRM) da Máquina Draco-PUC.

De maneira similar à utilizada na descrição dos cenários, foi definida a DTD para o LAL e gerada sua sintaxe Draco-PUC. Para facilitar a visualização dos artefatos, foram criadas folhas de estilo XSL, uma para cada artefato, que geram a representação em HTML levando em consideração as referências existentes e criando marcações que permitem a navegação entre os artefatos. A figura 32 mostra o código HTML gerado para o cenário *malfunction occurs* e sua visualização corresponde à apresentada na figura 29. A relação completa das DTDs e dos GRMs encontra-se nos apêndices A e B.

4.2.
Geração das Transformações de Rastreamento

O processo de geração das transformações de rastreamento é apresentado na figura 33. Inicialmente duas versões de um artefato são comparadas obtendo-se como resultado uma descrição das diferenças (em um arquivo .diff) e um conjunto de fatos observados sobre as duas versões. Sobre a descrição das diferenças é aplicado o transformador *HandleDiff*, gerando-se o conjunto de transformações equivalentes a estas diferenças. Os fatos observados sobre estas diferenças, em conjunto com os fatos sobre mudanças na configuração do sistema, servem como entrada para a aplicação do transformador *FindPlan*. Este transformador procura identificar qual o plano de uma biblioteca de planos que pode ser utilizado para descrever os fatos observados. Todas estas informações são então armazenadas pelo gerenciador de configuração.

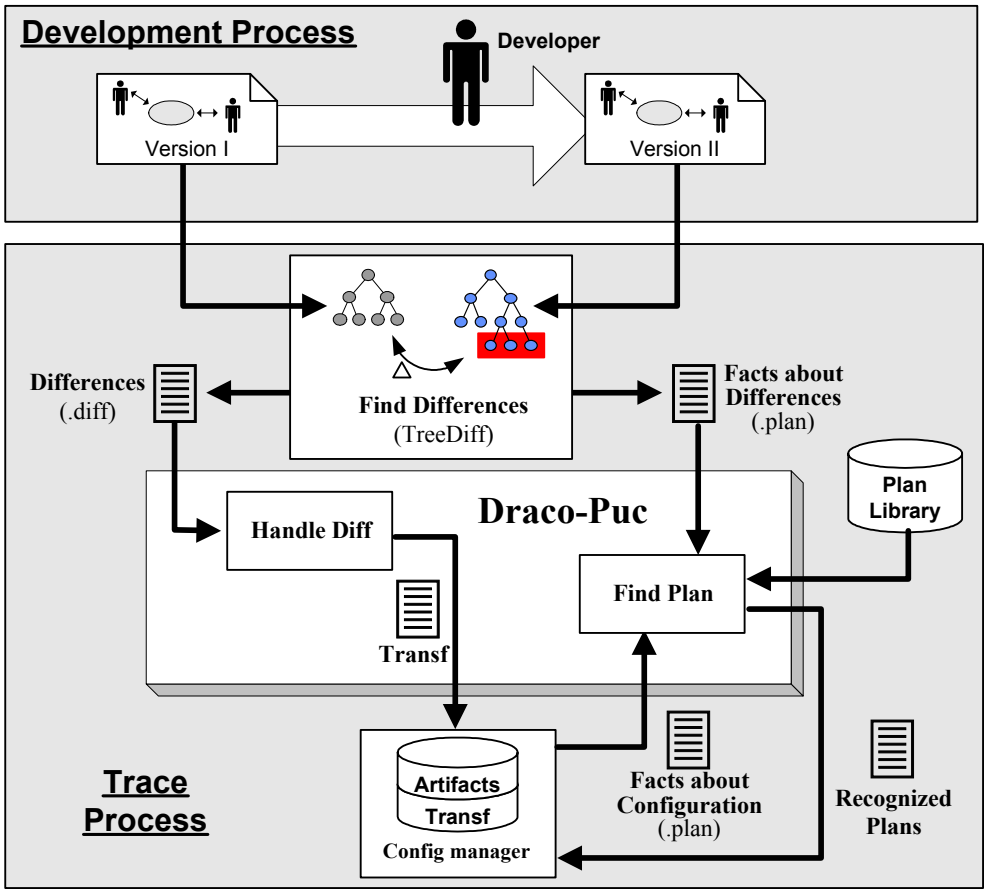

Figura 33 – Processo de Geração das Transformações de Rastreamento

4.2.1. Identificação de Diferenças

Foram identificadas e analisadas duas abordagens diferentes que poderiam ser utilizadas na identificação das diferenças entre dois artefatos. A primeira, através do armazenamento das ações realizadas pelo desenvolvedor ao longo da edição do artefato original, e a segunda através da identificação das diferenças a partir da comparação do artefato final em relação ao artefato original. Nossa escolha recaiu sobre a segunda alternativa pois, apesar de exigir uma carga maior de processamento, permite que o desenvolvedor utilize qualquer ferramenta existente na edição dos artefatos.

A abordagem escolhida baseia-se na utilização do algoritmo de identificação de diferenças entre árvores (*TreeDiff*). A aplicação deste algoritmo sobre as árvores correspondentes a dois artefatos resulta em uma descrição das diferenças encontradas, posteriormente utilizadas pela atividade de geração das transformações, e por um conjunto de fatos observados relacionados às diferenças encontradas, utilizado pela fase de reconhecimento de planos.

4.2.1.1. Algoritmo de Identificação de Diferenças (*TreeDiff*)

O algoritmo utilizado na identificação de diferenças entre árvores foi baseado no algoritmo proposto por Wang et al. [70]. Este algoritmo procura a maior subestrutura comum entre duas árvores e leva em consideração a ordem dos filhos de cada nó, apresentando como resultado um conjunto de operações de edição que deve ser aplicado na primeira árvore de maneira a obter-se a segunda. Cada operação (renomear, remover ou inserir um nó) possui um custo associado e o computo da subestrutura comum é realizado procurando-se minimizar o custo desta edição. A complexidade do algoritmo é dada por $O\left(\left|T_1\right| \times \left|T_2\right| \times \min\left(H_1, L_1\right) \times \min\left(H_2, L_2\right)\right)$, onde H_i, i = 1, 2 é o tamanho da árvore T_i e L_i, i = 1, 2 é o número de folhas da árvore T_i.

Este algoritmo pode ser utilizado para identificar tanto as similaridades quanto as diferenças entre duas árvores. A figura 34 apresenta um exemplo do resultado da aplicação do algoritmo para encontrar as diferenças entre as árvores A e B, na forma do seguinte conjunto de operações de edição sobre a árvore A: a

raiz deve ser renomeada para H ; os nós B e H devem ser removidos; e finalmente devem ser inseridos os novos nós B e I.

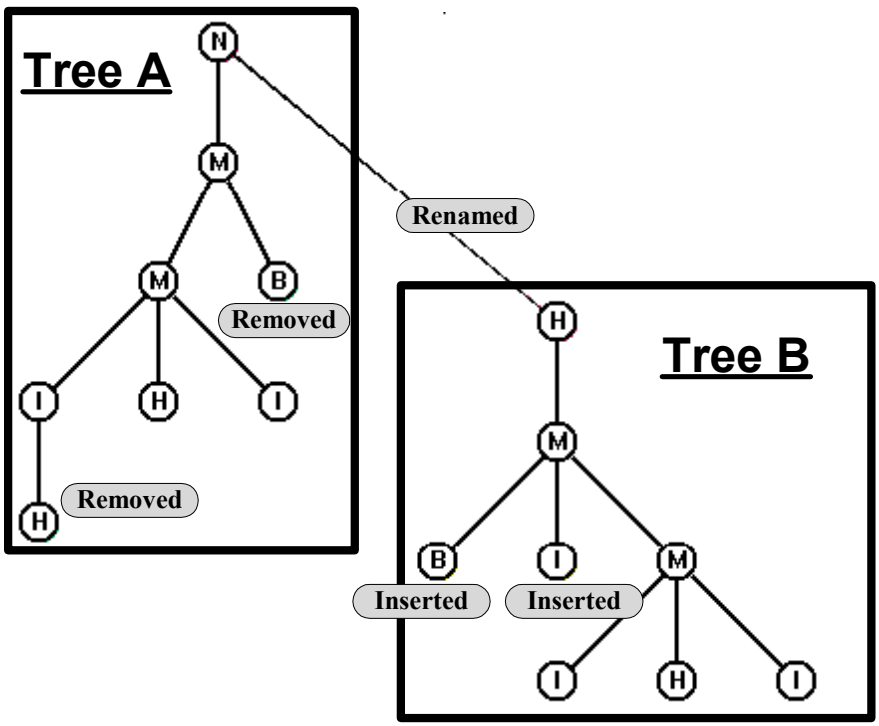

Figura 34 – Exemplo de aplicação do algoritmo TreeDiff

Inicialmente tentou-se aplicar o algoritmo sobre as árvores de sintaxe abstrata utilizados internamente pela Máquina Draco-PUC (DAST), porém estas árvores mantêm toda a informação sintática (as regras da gramática utilizadas quando da execução do parser) e desta forma apresentam uma série de informações dispensáveis. Optou-se então pela utilização da árvore criada diretamente pela leitura do arquivo XML e que apresenta a árvore correspondente de acordo com o modelo DOM (Document Object Model). A figura 35 apresenta um exemplo destas duas árvores para um mesmo elemento *Goal* de um cenário.

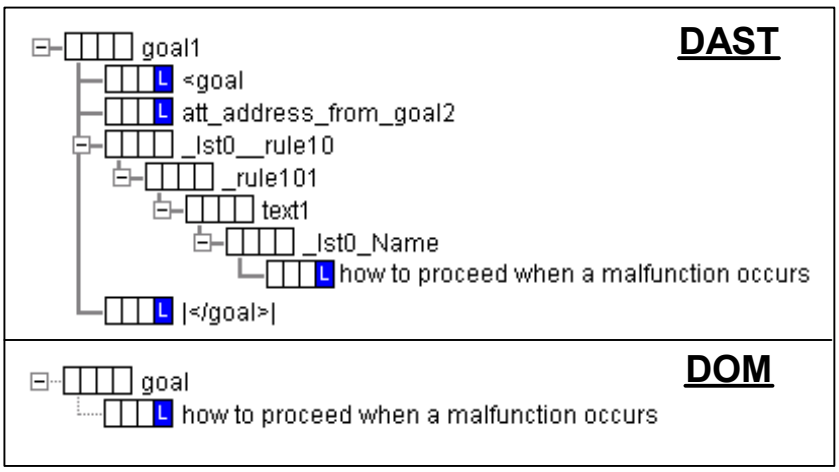

Figura 35 – Comparação DAST – DOM

A figura 36 mostra de que maneira são computadas as diferenças e similaridades entre dois arquivos XML. Inicialmente estes arquivos são lidos criando-se as duas árvores correspondentes, *DOMTree 1* e *DOMTree 2*. Sobre estas árvores é aplicado o algoritmo *TreeDiff*, obtendo-se como resultado o conjunto das operações a serem aplicadas na árvore *1* para obter a árvore *2*. Por fim, o conjunto de diferenças é processado de maneira a gerar uma lista de diferenças e um conjunto de fatos observados sobre as duas versões que possam ser utilizados na procura de qual plano caracteriza as ações realizadas pelo desenvolvedor. Este processamento das diferenças é apresentado nas seções 4.2.1.2 (descrição das diferenças) e 4.2.1.3 (fatos observados).

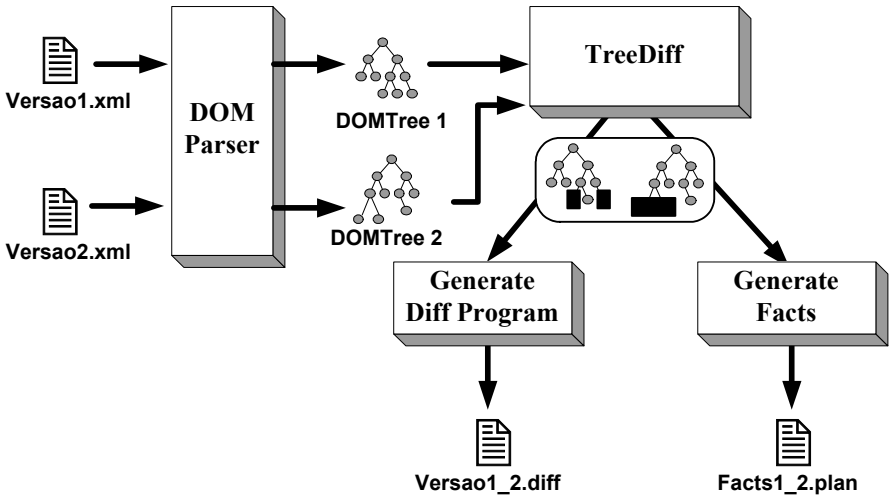

Figura 36 - Uso do Algoritmo TreeDiff

Foram realizadas diversas alterações no algoritmo original proposto por Wang et al. de maneira a otimizar sua utilização no mecanismo de rastreamento proposto. As principais alterações foram:

- O significado de cada nó no DOM é levado em consideração na comparação, ou seja, ao verificar se dois nós são iguais e qual a operação a ser realizada sobre o primeiro para obter-se o segundo, o algoritmo verifica se são nós que caracterizam um elemento, um atributo ou texto. Por exemplo, se os dois nós são do tipo texto, uma operação de renomear o nó tem um custo menor do que a inserção ou remoção do nó, ao passo que se o primeiro nó descreve um atributo e o segundo um elemento, o custo de renomear o nó é maior que o de removê-lo e inserir um novo nó.
- As folhas das árvores não são comparadas diretamente, e sim dentro do seu contexto definido pelos nós da sua hierarquia. Por exemplo, um texto

só é comparado com outro se os dois se referem ao mesmo elemento (não tem sentido comparar um texto que se refere ao objetivo do cenário com um que se refere ao contexto); e

- Os nós que representam referências (elemento *reference* da DTD de cenário e do domínio de cenários Draco-PUC) a outros elementos são tratados de maneira única levando em consideração o elemento que está sendo referenciado.

Para a implementação desta última característica foi necessário realizar uma extensão da notação originariamente definida para o LAL, de maneira a permitir a formalização de relacionamentos de composição e generalização dos seus termos, informações estas já implicitamente capturadas na definição de cada termo. Esta extensão foi necessária tendo em vista que estes relacionamentos devem ser levados em consideração no cálculo do custo de edição utilizado pelo algoritmo. Por exemplo, a troca de um termo do LAL por outro que é a especialização do primeiro deve ser menor do que a troca por um outro não relacionado.

Esta extensão foi implementada no elemento *Notion* da definição do termo do LAL através da explicitação do tipo do relacionamento (*ISA* e *COMPOSEDBY*) e dos termos relacionados. A definição do tipo correspondente na DTD do LAL pode ser vista no apêndice A.1 e a definição da linguagem do domínio LAL no Draco se encontra no apêndice B.1 . A figura 37 apresenta alguns exemplos de termos do LAL que utilizam esta notação.

A figura 40 mostra uma visualização do resultado da aplicação do algoritmo na identificação das diferenças entre as versões II (figura 38) e III (figura 39) do cenário *malfunction occurs*. Os ícones dos nós possuem dois retângulos: o da esquerda mostra informações sobre as diferenças (nó inserido, removido ou conteúdo renomeado) e o da direita sobre o tipo do nó (elemento, referência a um símbolo do LAL ou texto). Os nós marcados como *child* (metade inferior preenchido no retângulo da esquerda) são aqueles que não sofreram modificações, porém algum dos seus descendentes possui alguma diferença, como por exemplo o caso do ator pai da diferença marcada com o número 1 na árvore da esquerda.

Symbol: Sensor
Notion
Can measure some value
Has name, type, resolution, and reaction time
Behavior
Analog sensors respond in exponential time

Symbol: Motion detector
Notion
ISA sensor
It detects motion of a person or animal in its range
Behavior
Motion of a person in range triggers the value to 1
The system needs to handle malfunction of it

Symbol: Outdoor light sensor
Notion
ISA sensor
Behavior
It measures the illumination in a sphere perpendicular to it
The system needs to handle malfunction of it

Symbol: Control panel
Notion
COMPOSEDBY keyboard
COMPOSEDBY display
Behavior
The light scenes can be determined by using the control panel
A control panel is not available in the hallway section
A control panel is only available in the offices , computer lab and in the hardware lab
Should be easy and intuitive to use

Figura 37 – Exemplo do LAL com notação estendida

Goal	how to proceed when a malfunction occurs
Context	4th floor of building 32
Actor	Control system
Resource	Ceiling light groups
Resource	Control panel
Episode	informs facility manager of malfunction
Episode	inform user of malfunction
Episode	IF Hallway section is controllable neither automatically nor manually THEN Ceiling light groups have to be on
Episode	Control system supports the facility manager in finding the reasons of malfunction
Episode	store malfunction
Episode	facility manager can correct manually malfunction undetected by the Control system

Figura 38 - Cenário *malfunction occurs* – Versão II

Goal	how to proceed when a malfunction occurs
Context	4th floor of building 32
Actor	Central Control
Actor	outdoor light sensor
Actor	motion detector
Resource	Ceiling light groups
Resource	Control panel
Resource	last correct measurements from outdoor light sensor
Episode	Sensor informs Facility manager of malfunction
Episode	Sensor informs user of malfunction
Episode	Central Control supports the Facility manager in finding the reasons of malfunction
Episode	Central Control stores malfunction
Episode	Facility manager can correct manually malfunction undetected by Central Control

Figura 39 – Cenário *malfunction occurs* – Versão III

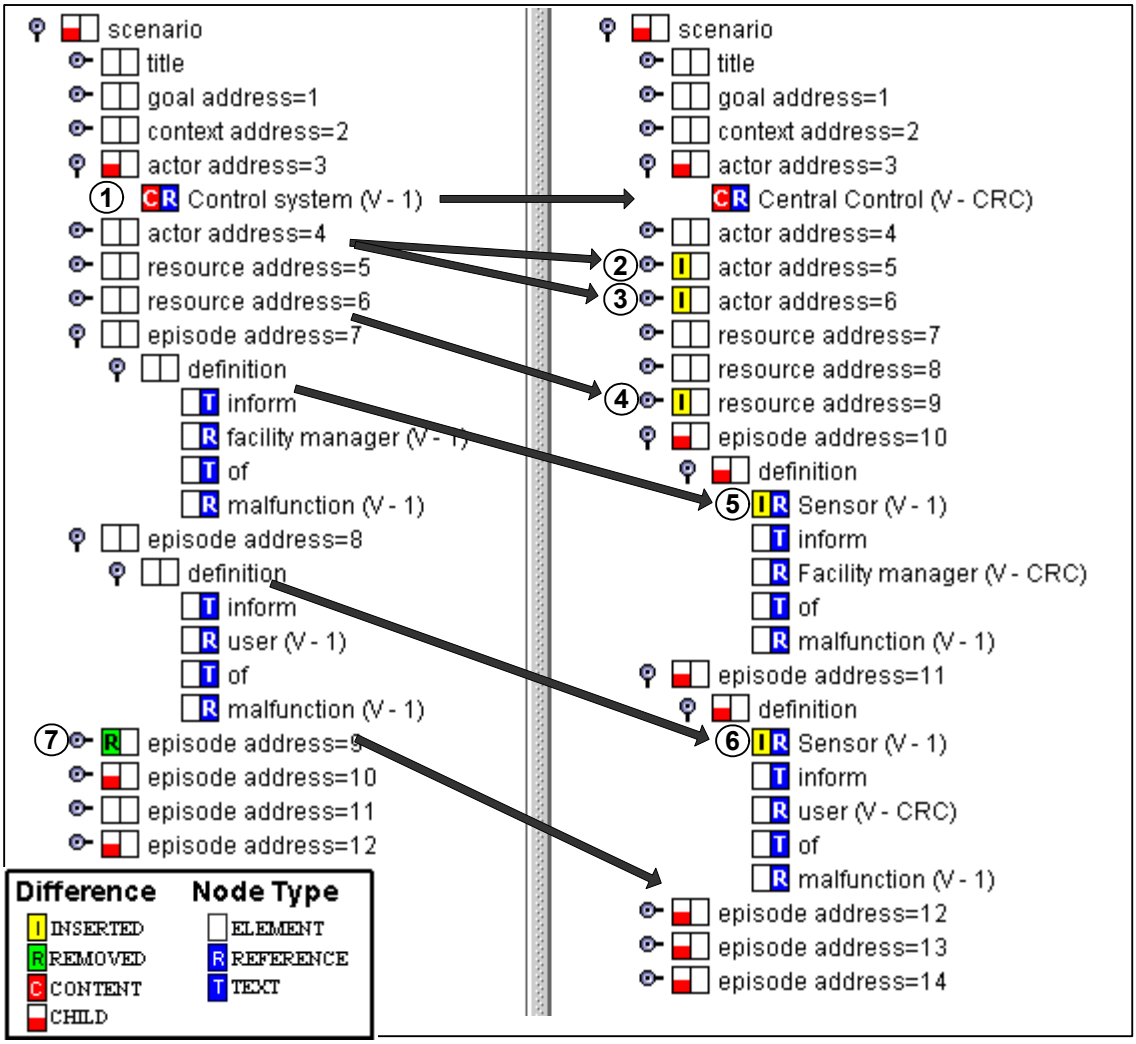

Figura 40 – Resultado da aplicação do algoritmo sobre os cenários *Malfunction occurs* Versão II (esquerda) e III (direita)

4.2.1.2. Descrição das Diferenças

A partir das diferenças encontradas pelo algoritmo *TreeDiff* é necessário acionar a Máquina Draco-PUC para que se gerem as transformações equivalentes a estas diferenças. Para tanto foram estudadas duas alternativas diferentes para a construção de um domínio de descrição de diferenças na Máquina Draco-PUC, a primeira envolvendo a geração de marcações embutidas na versão original e a segunda criando-se um novo artefato que contivesse apenas as diferenças. A figura 41 exemplifica estas alternativas para a descrição das diferenças entre o artefato 1 e o artefato 2. Na alternativa A é inserida uma marcação no local do artefato 1 correspondente a cada diferença encontrada, enquanto que na alternativa B é criado um novo arquivo para descrever as diferenças existentes.

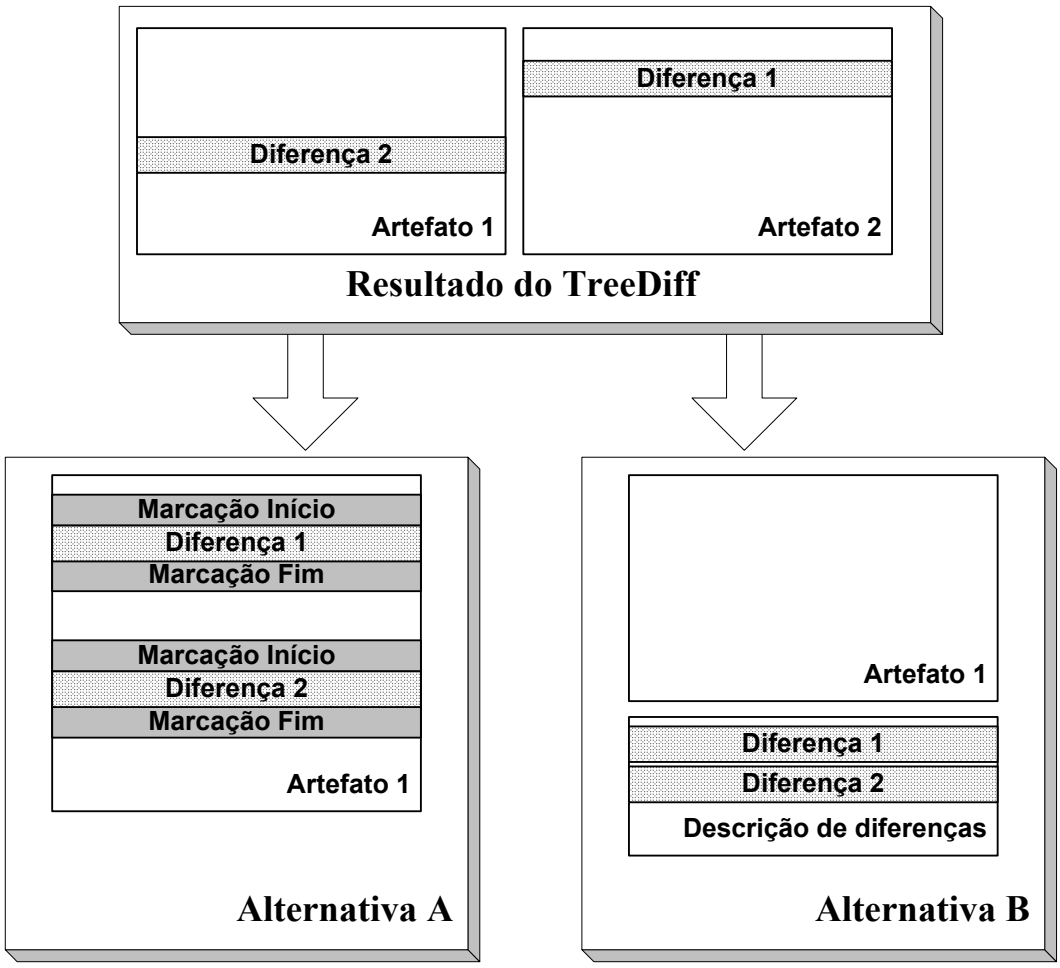

Figura 41 – Alternativas estudadas para a descrição das diferenças: A) embutir diferenças no artefato original; B) novo artefato com diferenças apenas

As principais vantagens da primeira alternativa são a facilidade de descrição do local de aplicação da diferença e a independência quanto à ordem de tratamento das diferenças. A segunda alternativa permite manter o artefato original inalterado e obter um maior isolamento entre as diferenças e o artefato original, facilitando o processamento posterior de geração das transformações. A principal desvantagem da segunda alternativa é a influência da ordem de aplicação das diferenças na descrição gerada. Por exemplo, se precisarmos inserir dois elementos irmãos a descrição será diferente caso se varie a ordem de aplicação das diferenças.

Da análise destas alternativas foi escolhida a segunda alternativa na definição do domínio de diferenças Draco-PUC. O problema correspondente a ordem de aplicação das diferenças na descrição gerada foi resolvido garantindo que a aplicação das transformações, correspondentes a cada descrição de

diferença, seja realizada na mesma ordem com que a descrição foi gerada e será apresentado posteriormente. A sintaxe deste domínio é mostrada na figura 42.

```
document            : diffDocument | embeddedStatement     ;
diffDocument        : diffElement* ;
diffElement         : replaceSpec |  insertSpec | removeSpec ;
replaceSpec         : 'replace' typeSpec 'from'  any  'to' any 'end' ;
insertSpec          : 'insert'  typeSpec  any where 'end' ;
removeSpec          : 'remove' typeSpec any where 'end' ;
typeSpec            : 'type' '=' whatChange ;
where               : 'in' any ;
embeddedStatement   : locater ;
locater             : 'start' | 'end' | 'here' ;
any                 : Text* ;
whatChange          : 'text' | 'reference'| 'title' | 'goal' | 'context' | 'actor' |
                      'resource' | 'episode' | 'definition';
String              : \"([^\"])*\" ;
String              : \'([^\'])*\' ;
Text                : [0-9a-zA-Z]* ;
```

Figura 42 – Domínio de diferenças na Máquina Draco-PUC (Diff)

Cada diferença encontrada deve ser mapeada diretamente para a construção do tipo *diffElement* correspondente: *replaceSpec* para mudança de conteúdo, *insertSpec* para inserção de um nó e *removeSpec* para a remoção de um nó. A construção *replaceSpec* especifica o tipo do nó que foi modificado, o conteúdo original do nó e o novo conteúdo. A construção *insertSpec* especifica o tipo do nó que foi inserido, o nó a ser inserido e o local de inserção. A construção *removeSpec* define o tipo do nó que foi removido, o nó a ser removido e a localização do mesmo.

O local de inserção ou remoção de um nó é definido através da cópia do trecho da árvore original acrescido de uma marcação do local exato de aplicação da diferença. Na marcação deste local é utilizado um poderoso recurso da Máquina Draco-PUC que é a possibilidade de utilizar descrições pertencentes a diversos domínios dentro de um mesmo programa. No exemplo da figura 43, as linha 1 e 5 são descrições pertencentes ao domínio de diferenças (*Diff*) e a linha 3 pertence ao domínio de cenários. Os caracteres **{{** , na linha 2, e **}}** , na linha 3, são utilizados para delimitar um bloco de código de um domínio diferente do atual.

A figura 44 mostra a descrição da diferença marcada com o número 5 na figura 40, que corresponde à inserção de uma referência para o símbolo *Sensor* do LAL no local marcado (trecho sublinhado) da definição de um episódio. Esta descrição foi gerada a partir do resultado da aplicação do *TreeDiff* e utiliza tanto a linguagem do domínio de diferenças (*Diff*), na definição do tipo da diferença,

quanto a linguagem do domínio de cenários (*Cen*), na especificação dos elementos inseridos e do local de inserção.

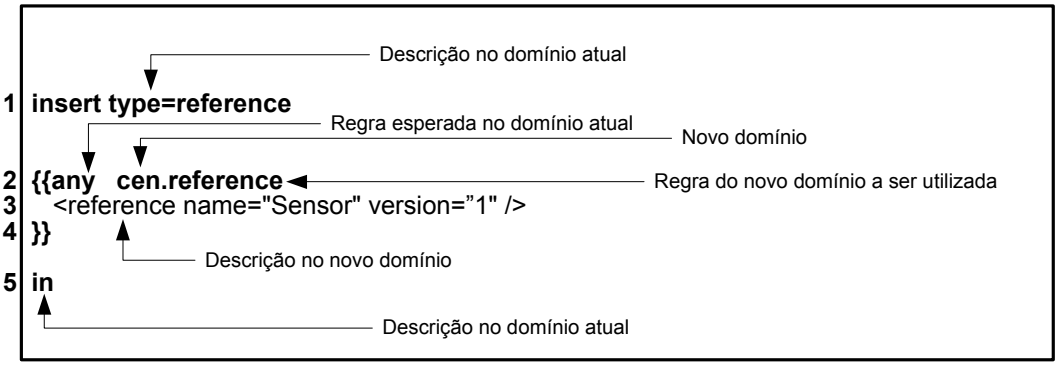

Figura 43 – Uso de descrições de diversos domínios na Máquina Draco-PUC

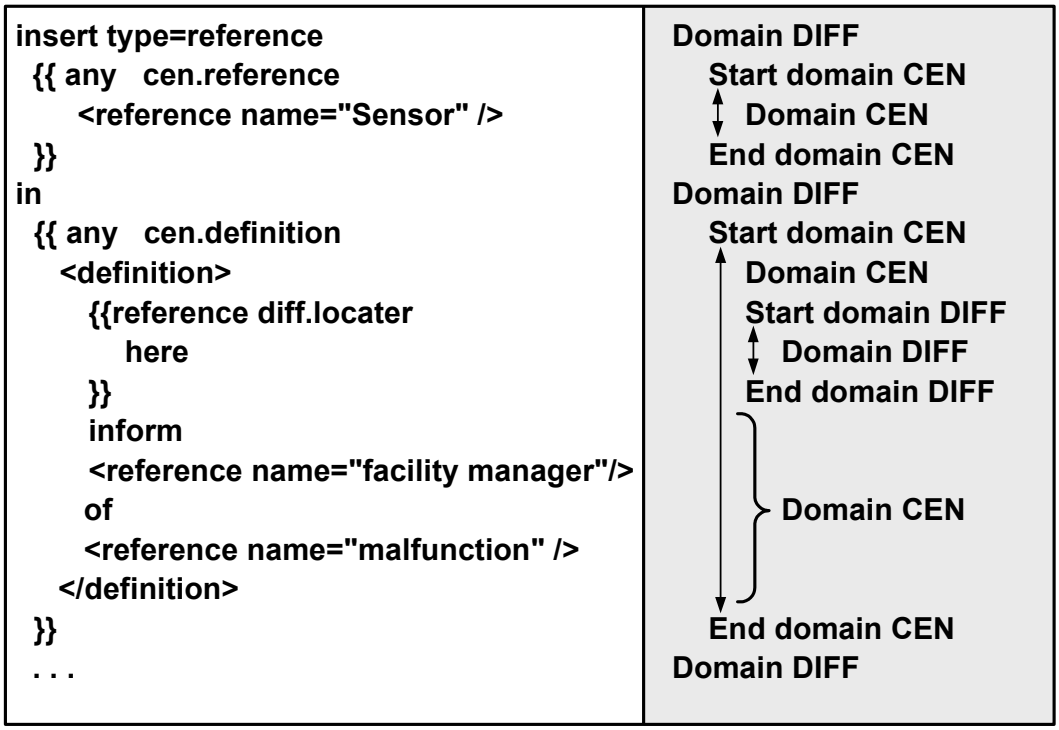

Figura 44 – Exemplo de descrição de diferença no domínio *Diff*. A parte direita da figura mostra os diversos domínios utilizados.

Conforme citado anteriormente, a descrição das diferenças é influenciada pela ordem com que as diferenças são tratadas para gerar sua descrição. Para solucionar este problema torna-se necessário garantir que a ordem de geração da descrição das diferenças e a ordem de aplicação das transformações correspondentes seja a mesma. Esta ordem foi mantida através da geração de um conjunto de transformações para cada diferença. Este conjunto é composto por uma única transformação e é aplicado na ordem com que foi inserido no arquivo de transformações.

Outra questão que necessitava ser tratada era que as diferenças anteriormente tratadas tinham que ser levadas em consideração na geração da descrição de uma nova diferença. A solução adotada foi, para cada diferença identificada, aplicar a diferença na árvore original imediatamente após a sua descrição ter sido gerada, possibilitando que todas as diferenças subseqüentes levassem em consideração as diferenças já tratadas.

Inicialmente a geração das descrições foi realizada na ordem com que as diferenças eram identificadas, independentemente do tipo da diferença. Esta forma de geração mostrou-se inadequada pois introduzia alguns erros quando da inserção de um novo texto que tivesse como um dos seus irmãos um outro texto, ou quando da remoção de um nó cujos irmãos da esquerda e da direita fossem ambos do tipo texto. Nestes casos ocorria a junção dos textos em um único nó o que prejudicava a geração das descrições subseqüentes. A solução adotada foi realizar a geração das descrições de acordo com o tipo da diferença, tratando primeiramente as mudanças do conteúdo de um nó, depois as inserções de novos nós e por último as remoções de nós da árvore original. Desta maneira evitou-se a geração de nós do tipo texto adjacentes e o problema foi contornado.

O último ponto que necessitava ser tratado era quanto ao nível de descrição da diferença. Este nível relaciona-se a granularidade do elemento considerado na diferença: quanto mais baixo o nível maior será a granularidade do elemento. Em seu nível mais baixo (o nível 0), a diferença relaciona-se apenas ao nó que contém a diferença. Cada nível superior é formado pelo pai da diferença no nível imediatamente inferior. Os níveis para a diferença de número 1, mostrada na figura 40 e que representa a troca de uma referência por outra, são mostrados na figura 45. Neste caso, a descrição da diferença no nível 0 seria a troca de uma referência (*Control System*) pela outra (*Central Control*). No nível 1 a descrição envolveria a troca de toda a definição e no nível 2 a troca de todo o episódio. Quanto mais baixo o nível da descrição menor será a quantidade de informações necessárias na descrição e mais simples serão as transformações geradas. O problema da utilização dos níveis mais baixos reside na possibilidade de ocorrer a ativação da transformação em outros pontos do artefato e não somente naqueles que necessitam ser modificados. A utilização do nível 0 no exemplo da figura 45 irá gerar uma transformação que trocará todas as referências existentes para o termo *Control System* e não apenas a existente na definição do episódio 10 do

artefato original. O mecanismo permite a configuração do nível de geração das descrições em função do tipo de diferença encontrada (renomear, inserir e remover) e do tipo do nó (texto, referência ou elemento).

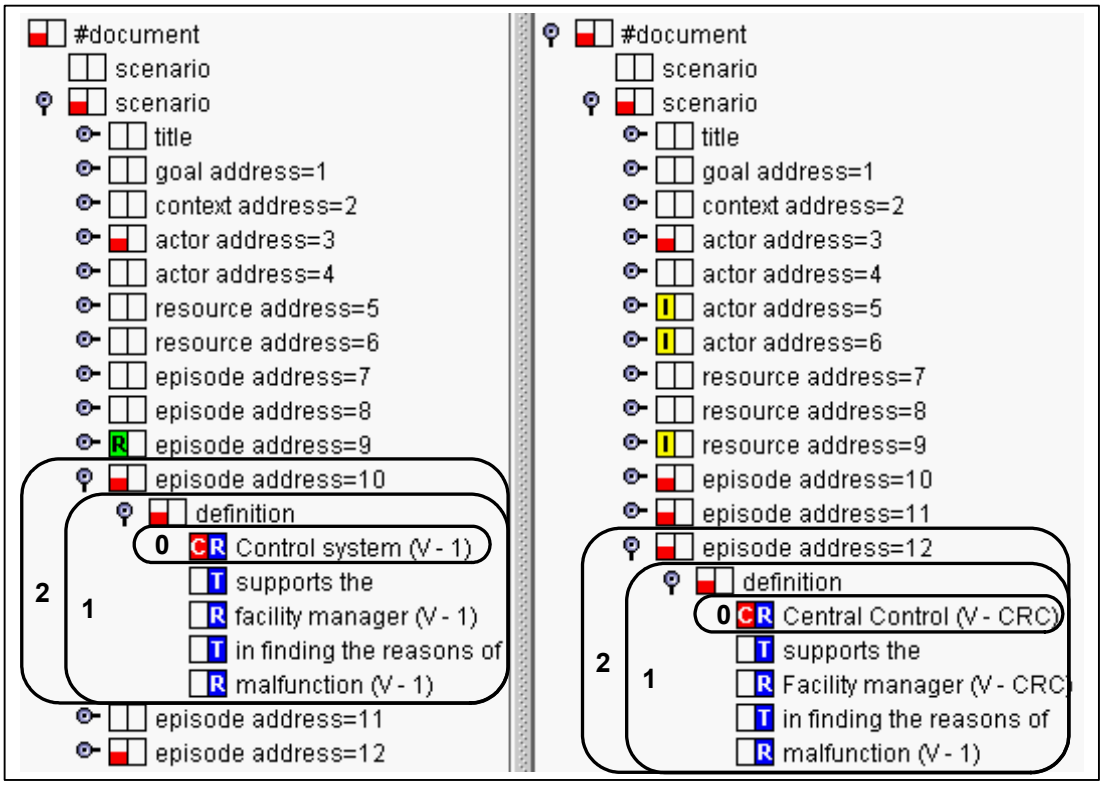

Figura 45 - Exemplo dos níveis de descrição de uma diferença

A geração das descrições para as diferenças mostradas na figura 40 é realizada da seguinte forma. Inicialmente é tratada a diferença de número 1 que corresponde a troca de uma referência por outra. Adotando-se o nível de granularidade 1, a descrição gerada no domínio Draco-PUC correspondente é a seguinte:

```
replace type=actor
    from
        {{any cen.actor
            <actor address="3">
              <reference name="Control system" version="1"/>
            </actor>
        }}
    to
        {{any cen.actor
            <actor address="3">
                <reference name="Central Control" version="CRC"/>
            </actor>
        }}
end
```

Após a geração desta descrição, a diferença é aplicada sobre a árvore original levando a situação apresentada na figura 46.

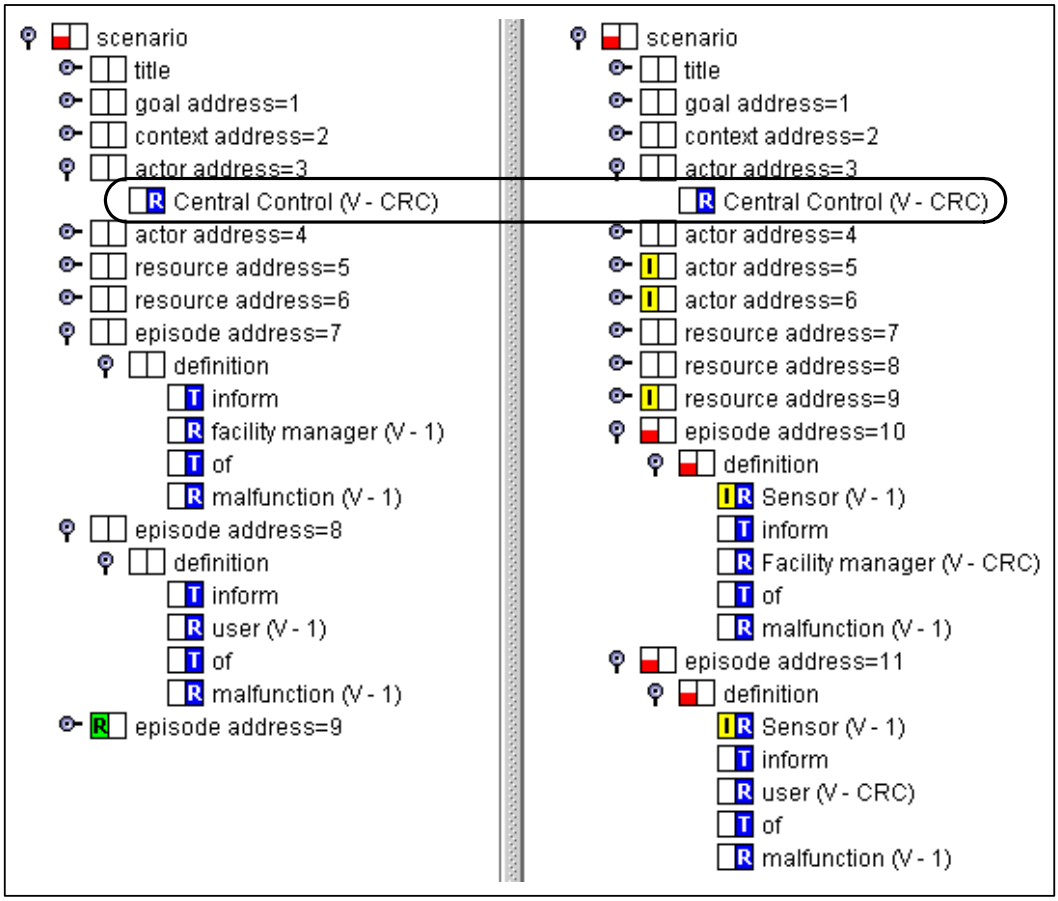

Figura 46 – Geração do programa representativo das mudanças – Situação do cenários *Malfunction occurs* da versão I (esquerda) e II (direita) após a primeira interação.

O processo se repete para cada nova diferença. Um exemplo de tratamento de uma inserção é o realizado para a diferença de número 4 na figura 40, que prevê a inserção de um novo elemento *resource*, gerando a descrição apresentada a seguir e fazendo com que, após este tratamento, as árvores apresentem a situação mostrada na figura 47.

```
insert type=resource
    {{any cen.resource
        <resource address="9">
            last correct measurements from
            <reference name="outdoor light sensor" version="CRC"/>
        </resource>
    }}
in
  {{any cen.scenario
        <scenario>
            [[title aTitle]]
            [[goal aGoal]]
            [[context aContext]]
            [[actor anActor]]
            [[actor anActor1]]
            [[resource aResource]]
            <resource address="6">
                <reference name="Control panel" version="1"/>
```

```
            </resource>
            {{resource diffaux.locater here}}      //  local da inserção
            <episode address="7">
                <definition>
                    inform
                    <reference name="facility manager" version="1"/>of
                    <reference name="malfunction" version="1"/>
                </definition>
            </episode>
            [[episode anEpisode1]]
            [[episode anEpisode2]]
            [[episode anEpisode3]]
            [[episode anEpisode4]]
            [[episode anEpisode5]]
        </scenario>
    }}
end
```

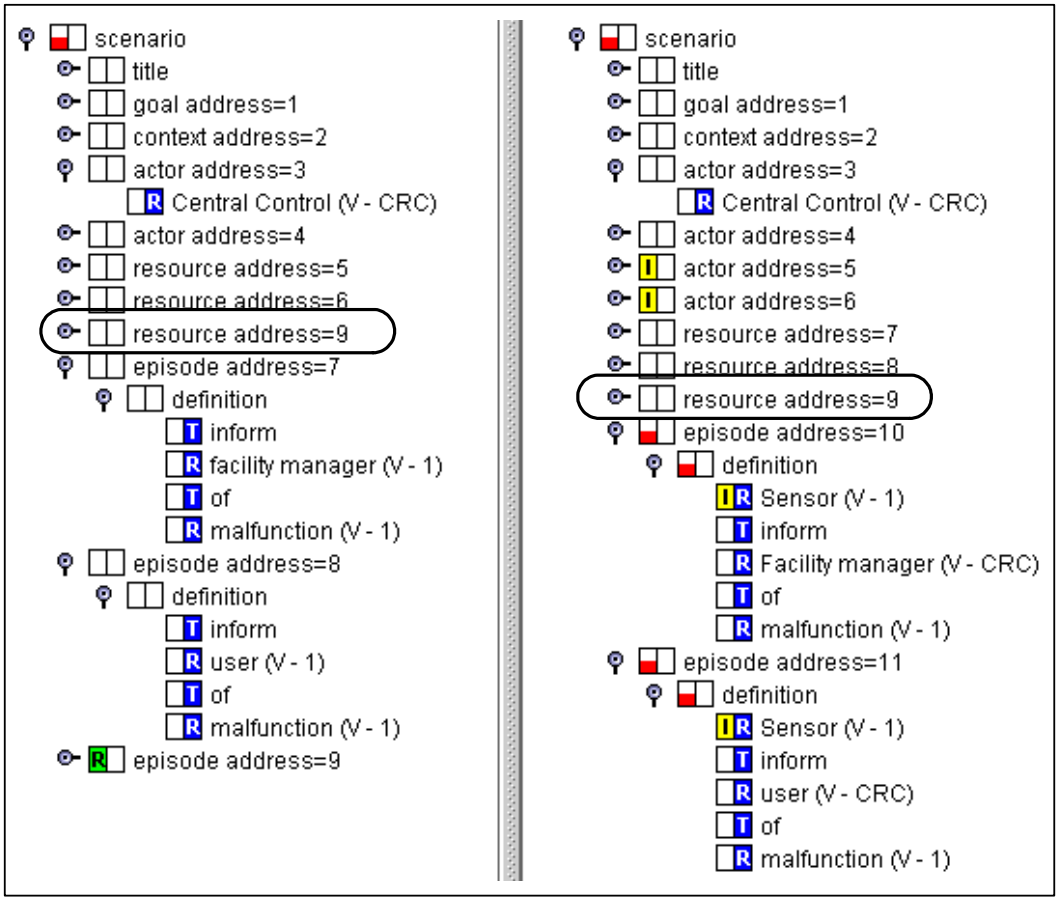

Figura 47 - Geração do programa representativo das mudanças – Situação do cenário *Malfunction occurs* da versão I (esquerda) e II (direita) após a segunda interação.

A construção delimitada por **[[** e **]]** é utilizada pela Máquina Draco-PUC para especificar uma variável. No momento da aplicação da transformação correspondente a esta diferença, uma variável pode ser amarrada a qualquer ponto da DAST que tenha o mesmo tipo definido para a variável, neste caso a existência de 5 episódios (*anEpisode1* até *anEpisode5*). A utilização de variáveis se faz necessária para evitar que na definição do local de aplicação tenha-se que repetir

todo o cenário. A regra que foi definida para a construção da descrição é que devem ser utilizadas variáveis para todos os nós do mesmo nível da marcação de local, com exceção do seu irmão esquerdo e direito, que devem ser colocados de forma completa de maneira a definir o local exato de aplicação.

4.2.1.3. Geração dos Fatos Observados sobre as Diferenças

O outro resultado da atividade de identificação de diferenças que deve ser produzido é uma lista de fatos observados sobre as diferenças identificadas pelo algoritmo *TreeDiff*. Por se tratarem de observações que não descrevem exatamente qual a diferença e sim o seu significado, a ordem com que os fatos aparecem nesta lista, ao contrário da descrição das diferenças, não tem qualquer influência no mecanismo. Estes fatos são utilizados pela técnica de reconhecimento de planos, que será apresentada na seção 4.2.3, que tem como objetivo reconhecer quais dos planos existentes em uma biblioteca de planos podem ser utilizados para explicar os fatos observados. Em conseqüência, a definição de quais os fatos observados que devem ser gerados a partir das diferenças é função da biblioteca de planos que será utilizada.

A descrição dos fatos deve seguir a sintaxe do domínio de reconhecimento de planos (*Plan*) da Máquina Draco-PUC que será apresentado na seção 4.2.3. A figura 48 mostra a parte deste domínio que especifica a sintaxe a ser utilizada na descrição dos fatos observados. Cada fato observado (*observedEvent*) possui um nome, que identifica unicamente este fato dentro da biblioteca de planos correspondente, e uma descrição da observação (*observation*) que contextualiza este fato. A observação é formada por uma lista de argumentos e por um valor opcional (*value*) que permite que se quantifique o fato observado.

```
instanceProgram   : 'Observations'
                    'begin'
                       observedEventList
                    'end' ;
observedEventList : observedEvent* ;
observedEvent     : eventName observation ;
observation       : argList value ;
argList           : '('  argValue**','  ')' ;
value             : '[' String ']'   |   ;
argValue          : String version | '*'  ;
version           : ':' String  |  ;
```

Figura 48 – Domínio de Planos Draco-PUC / Descrição de fatos observados

A figura 49 apresenta alguns exemplos de fatos observados que podem ser gerados a partir da diferença de mudança do conteúdo de um ator (troca de referencia) encontrada entre as versões II e III do cenário *malfunction occurs* (diferença número 1 da figura 40). É importante ressaltar que o significado dado aos fatos observados da figura é meramente ilustrativo, tendo sido utilizado para mostrar a flexibilidade existente na definição de sua sintaxe. Este significado está intrinsecamente relacionado à utilização dos fatos observados na definição dos planos da biblioteca de planos correspondente. Um conjunto concreto de definições de fatos observados pode ser visto na seção 5.1 que mostra a utilização do mecanismo de rastreamento na definição de um sistema de rastreamento para o desenvolvimento de software baseado em cenários.

Fato Observado	Significado
scenarioContentChanged (“malfunction occurs”: “2”,“actor”)[“1.0”]	Informa que houve uma mudança no conteúdo do cenário *malfunction occurs* na versão 2 relativo a mudança de um ator. O valor deste fato é igual a 1,0.
scenarioContentChanged(“malfunction occurs”:“2”,“actor”,“3”)	Informa que houve uma mudança no conteúdo do cenário *malfunction occurs* na versão 2. Esta mudança é relativa ao ator de número 3
referenceChanged(“Control System”:“1”,“Central Control”:“CRC”)	Mudança de uma referência por outra

Figura 49 – Exemplos de fatos observados sobre diferenças entre artefatos

4.2.2. Geração das Transformações

A geração das transformações equivalentes às diferenças encontradas entre os artefatos é realizada através da aplicação do transformador *HandleDiff* sobre a descrição das diferenças produzidas na atividade anterior. A figura 50 apresenta este transformador. Cada descrição de diferença existente é reconhecida pela transformação correspondente ao seu tipo. Esta transformação gera um novo conjunto de transformações composto por uma única transformação que, quando for aplicada ao artefato original, reproduz a diferença correspondente. A aplicação deste transformador mantém a mesma ordem pré-existente das descrições no arquivo original nos conjuntos de transformações gerados.

Transformer handleDiff		
Set of Transforms HandleDifferences		
Transform findInsertion		
	LHS	insert type = [[whatChange wc]] [[any what]] in [[any where]] end
	Pre-Apply	*1) Cria um novo SetOfTransforms* *2) Cria uma nova transform* *LHS: where retirando a marcação do local* *RHS: where trocando a marcação do local por what* *3) Salvar o novo SetOfTransforms*
Transform findRemove		
	LHS	remove type = [[whatChange wc]] [[any what]] in [[any where]] end
	Pre-Apply	*1) Cria um novo SetOfTransforms* *2) Cria uma nova transform* *LHS: where retirando a marcação do local* *RHS: where retirando what a partir da marcação do local* *3) Salvar o novo SetOfTransforms*
Transform findContentChange		
	LHS	replace type = [[whatChange wc]] from [[any original]] to [[any new]] end
	Pre-Apply	*1) Cria um novo SetOfTransforms* *2) Cria uma nova transform* *LHS: original* *RHS: new* *3) Salvar o novo SetOfTransforms*

Figura 50 – Transformador de Geração das Transformações de Rastreamento

A transformação *findInsertion* é acionada cada vez que for encontrada uma declaração de inserção na descrição das diferenças. Esta transformação cria um novo conjunto de transformações e insere uma nova transformação neste conjunto. A transformação é criada a partir da declaração de inserção da seguinte maneira: o LHS é formado retirando-se a marcação do local de inserção da cláusula *where* e o RHS é formado pela cláusula *where* tendo a marcação do local sido substituída pela cláusula *what*. Por exemplo, a declaração de inserção de uma nova referência:

```
insert type=reference
    {{any cen.reference                          // cláusula what
        <reference name="Sensor" version="1"/>
    }}
in
    {{any cen.definition                         // cláusula where
        <definition>
            {{reference diffaux.locater here}}   // marcação do local
            inform
            <reference name="facility manager" version="1"/>
            of
            <reference name="malfunction" version="1"/>
        </definition>
    }}
end
```

irá gerar o seguinte conjunto de transformações:

```
Set Of Transforms SOT1
    Trigger: external
```

```
Transform TFM1
 Lhs: {{any cen.definition
    <definition>
        inform
        <reference name="facility manager" version="1"/>
        of
        <reference name="malfunction" version="1"/>
    </definition>
}}
 Rhs: {{any cen.definition
    <definition>
        <reference name="Sensor" version="1"/>        // elemento inserido
        inform
        <reference name="facility manager" version="1"/>
        of
        <reference name="malfunction" version="1"/>
    </definition>
}}
```

A transformação equivalente a diferença relativa a remoção de um nó é gerada pela transformação *findRemove*. O LHS da transformação é gerado retirando-se as marcações de início (*start*) e término (*end*) do local de remoção. O RHS é gerado removendo todos os nós existentes entre as marcações de início e término. Por exemplo, para a declaração de remoção de uma referência da definição de um episódio:

```
remove type=reference
    {{any cen.reference                              // cláusula what
        <reference label="system" name="Control system" version="1"/>
    }}
in
  {{any cen.definition                               // cláusula where
        <definition>
            {{reference diffaux.locater start}}      // marcação de início de remoção
                <reference label="system" name="Control system" version="1"/>
            {{reference diffaux.locater end}}        // marcação de término de remoção
            Find reason for <reference name="malfunction" version="1"/>
        </definition>
    }}
end
```

será gerado o conjunto de transformações:

```
Set Of Transforms SOT2
    Trigger: external
    Transform TFM1
     Lhs:  {{any cen.definition
        <definition>
            <reference label="system" name="Control system" version="1"/>
            Find reason for <reference name="malfunction" version="1"/>
        </definition>
    }}
     Rhs: {{any cen.definition
        <definition>
            Find reason for <reference name="malfunction" version="1"/>
        </definition>
    }}
```

Finalmente, a geração da transformação correspondente a uma mudança do conteúdo de um nó (renomear) é gerada diretamente através da cláusula *original* que formará o LHS e da cláusula *new* que será o RHS. Desta forma, para a descrição da diferença:

```
replace type=actor
    from
        {{any cen.actor
            <actor address="4">
              <reference name="4th floor of building 32" version="1"/>
            </actor>
        }}
    to
      {{any cen.actor
        <actor address="4">
          <reference name="facility manager" version="1"/>
        </actor>
    }}
end
```

será gerado o conjunto de transformações:

```
Set Of Transforms SOT2
    Trigger: external
    Transform TFM1
     Lhs:  {{any cen.actor
        <actor address="4">
          <reference name="4th floor of building 32" version="1"/>
        </actor>
    }}
    Rhs:  {{any cen.actor
        <actor address="4">
          <reference name="facility manager" version="1"/>
        </actor>
    }}
```

4.2.3. Reconhecimento de Planos

O reconhecimento de planos pode ser definido como a tarefa realizada com o objetivo de determinar o melhor conjunto hierárquico de planos que explica uma série de fatos observados [71]. Esta tarefa geralmente é limitada a um domínio especifico que limita e restringe os fatos que possam vir a serem observados.

A implementação da técnica de reconhecimento de planos realizada neste trabalho foi baseada no algoritmo geral apresentado por Kauts [72] que procura reconstruir um plano preexistente que contenha os fatos observados. Nossa implementação é baseada em um domínio Draco-PUC de planos (Plan), constituído de uma linguagem para descrição de planos e de fatos observados e de

um transformador que executa a construção de planos a partir dos fatos observados.

A definição da linguagem para descrição de planos e de fatos observados é apresentada na figura 51. Esta linguagem é utilizada para descrever 3 tipos diferentes de programas: a biblioteca de planos (elemento *planLibrary* da linguagem), que define todos os fatos que podem ser observados e os planos que podem ser selecionados para explicar estes fatos; a descrição dos fatos efetivamente observados (elemento *instanceProgram*) a partir dos quais se deseja realizar o reconhecimento do plano correspondente; e a descrição dos planos reconhecidos (elemento *planRecognition*), gerada pela aplicação da técnica e que contém os planos que explicam o conjunto de fatos observados e a sua justificativa, expressa através de uma hierarquia de eventos inferidos construída a partir dos fatos observados.


```
program : planLibrary? instanceProgram? planRecognition? ;
planLibrary : planLibraryImport | planLibraryDefinition ;
planLibraryImport : 'loadLibrary' String '.' ;
planLibraryDefinition :  'PlanLibrary' 'begin' eventDeclaration*  'end' ;
eventDeclaration: eventIdentification parListdecl eventAttribute declBody ;
eventIdentification :   'Event'   eventName ;
parListdecl : '('  parDeclaration**','  ')' ;
parDeclaration : Name | previusVersion | nextVersion              | any;
previusVersion : Name '-' ;
nextVersion : Name '+';
eventAttribute : 'is'   'EndEvent' |    ;
any : '*' ;
declBody : 'begin' specializationRule? decomposition? 'end' | ';'  ;
specializationRule : 'isa'   eventName   ;
decomposition : 'composedBy'  decompositionStatement+  ;
decompositionStatement : eventName parListdecl decompositionRule ';' ;
decompositionRule : 'by'   ruleName parameters ;
parameters :   'with'   parameterPair++','  |  ;
parameterPair : Name   '='   String   ;
ruleName : Name ;
eventName : Name ;
instanceProgram :  'Observations'  'begin'  observedEventList 'end' ;
observedEventList : observedEvent* ;
observedEvent :   eventName observation ;
observation : argList value ;
argList : '('  argValue**','  ')' ;
value : '[' String ']' |  ;
argValue : String version | '*' ;
version : ':' Number | ;
planRecognition :  'Recognition' 'begin' topInferredEvent* 'end' ;
topInferredEvent :   inferredEvent '.'            ;
inferredEvent : eventName   argList   value   '{' rationaleStatement+ '}' ;
anyEvent : inferredEvent | observedEvent ;
rationaleStatement :  rationale ;
rationale : observedEvent | 'isa'   anyEvent  | inferredPartOfEvent++','   ;
inferredPartOfEvent : ruleName ':' anyEvent ;
```

Figura 51 – Linguagem de descrição de planos

A figura 52 mostra um exemplo parcial de descrição de uma biblioteca de planos neste domínio. Os eventos definidos como *EndEvent* caracterizam os planos que podem ser selecionados para explicar um subconjunto dos fatos observados (o evento *fusionOperation* da figura 52), ou seja, representam o resultado da aplicação da técnica. Os demais eventos são utilizados como passos intermediários inferidos a partir de outros eventos ou fatos que foram diretamente observados.

Cada evento definido na biblioteca possui um nome, um conjunto de parâmetros e pode possuir ainda dois tipos de relacionamentos: de decomposição (*composedBy*) e de generalização(*isa*). O nome identifica unicamente o evento e os parâmetros são utilizados para contextualizar a ocorrência do evento dentro de uma determinada situação de uso da técnica.

O relacionamento de generalização indica que toda vez que o evento declarado for inferido, o evento que representa a generalização deste automaticamente é inferido também. Neste caso a declaração dos parâmetros destes eventos devem ser os mesmos. Na figura 52 os eventos *episodeAdded* e *episodeRemoved* são especializações do evento *episodeModification* e, portanto, possuem os mesmos parâmetros e toda vez que forem inferidos o evento *episodeModification* também o será.

A decomposição representa um relacionamento todo-parte e indica os eventos (partes) que contribuem para que o evento declarado (todo) seja inferido. Cada decomposição deve especificar o nome do evento parte, os argumentos do evento todo que correspondem aos parâmetros do evento parte, um nome para a regra de composição e um peso associado. O evento definido como parte deve estar declarado em algum outro local da biblioteca. O peso quantifica a importância de cada evento parte na construção do evento todo em relação aos demais eventos partes.

A definição dos parâmetros e dos argumentos influi na forma com que o evento todo será inferido a partir do momento em que o evento parte tiver sido inferido. A geração do evento todo obedece as seguintes regras:

1) Os parâmetros do evento parte são replicados no evento todo de acordo com a correspondência dos parâmetros na declaração. Por exemplo, quando o evento *episodeRemoved* tiver sido inferido com os parâmetros (cenário X, episódio Y) os parâmetros do evento todo gerado *episodeMoved* serão (cenário X,

* , episódio Y), onde o asterístico (*) representa um coringa, ou seja, o evento parte não especifica qual deve ser o parâmetro e este pode ser futuramente preenchido com qualquer outro evento.

2) O uso do asterístico na declaração do evento parte significa que aquele parâmetro pode ser qualquer um, ou seja, este parâmetro é descartado quando da geração do evento todo.

3) Os símbolos de mais (+) e menos (-) representam, respectivamente, a próxima versão de um artefato e a versão anterior. No exemplo da figura 52 uma operação de fusão entre dois cenários A e B é inferida quando ocorrer o evento *complementRelationship* para a versão anterior do cenário A e a versão atual do cenário B. Por exemplo, na ocorrência do evento *complementRelationship* entre os cenários X na versão 5 e Y na versão 6 será gerado o evento *fusionOperation* para os cenários X na versão 6 e Y na versão 6.

```
01 PlanLibrary
02 begin
03   Event episodeModification(A,B);          // A:scenario  B:episode
04   Event complementRelationship(A,B);       // A:scenario  B:scenario
05   Event scenarioRemoved(A);                // A:scenario
06   Event episodeRemoved(A,B)                // A:scenario  B:episode
07     begin
08       isa episodeModification
09     end
11   Event episodeAdded(A,B)                  // A:scenario  B:episode
12     begin
13       isa episodeModification
14     end
15   Event episodeMoved(A,B,C)                // A:scenario  B:episode
16   begin
17     composedBy
18       episodeRemoved(A,C) by source with weight = '1.0';
19       episodeAdded(B,C) by target with weight = '1.0';
20   end
21   Event fusionOperation(A,B)  is EndEvent  // A:scenario  B:scenario
22   begin
23     composedBy
24       scenarioRemoved(B) by r1 with weight = '1.0';
25       complementRelationship(A-,B) by r2 with weight = '0.5';
26       episodeMoved(B,A,*) by r3 with weight = '1.0';
27   end
28 end
```

Figura 52 – Exemplo de definição de biblioteca de planos

A sintaxe utilizada na descrição dos fatos observados foi anteriormente apresentada na seção 4.2.1.3 e está intimamente relacionada à biblioteca de planos que será utilizada no reconhecimento. A figura 53 apresenta um exemplo de descrição de fatos observados para a biblioteca de planos da figura 52.

```
01 Observations
02 begin
03   complementRelationship("malfunction":"1" , "malfunction occurs":"2" ) [0.765] ;
04   scenarioRemoved("malfunction":"1");
05   episodeRemoved("malfunction":"1" , "episode3");
06   episodeAdded("malfunction occurs":"2" , "episode3");
07 end
```

Figura 53 – Exemplo de descrição de fatos observados

Os fatos observados podem ser gerados por quaisquer componentes do mecanismo de rastreamento. As principais fontes de fatos observados são o gerenciador de configuração, o algoritmo de identificação de diferenças (*TreeDiff*), a própria aplicação anterior da técnica de reconhecimento de planos ou ainda a aplicação de transformações na Máquina Draco-PUC.

O gerenciador de configuração é responsável pela geração dos fatos relacionados às mudanças de configuração do sistema, tais como a inserção ou remoção de artefatos. Um exemplo de fato gerado por este gerente é o que aparece na linha 4 da figura 53, que representa a remoção do cenário *malfunction* – Versão I da configuração que está sendo analisada.

O algoritmo *TreeDiff* gera os fatos relacionados às diferenças entre os artefatos analisados, como os fatos das linhas 5 e 6 da figura 53 que mostram, respectivamente, que o episódio 3 do cenário *malfunction* – Versão I foi removido e que foi inserido o episódio 3 no cenário *malfunction occurs* – Versão II.

Fatos observados podem ser criados pela aplicação anterior da técnica de reconhecimento de planos, como o mostrado na linha 3 da figura 53, que informa que os cenários *malfunction* – Versão I e *malfunction occurs* – Versão I possuem um relacionamento de complemento. O valor mostrado (0,765) representa um nível de certeza quanto a existência do relacionamento.

A aplicação de transformações é uma importante fonte de fatos observados. Transformações podem ser escritas para reconhecer determinados padrões que caracterizam a ocorrência de um fato observado. No ponto de controle de *Pré-Apply* da transformação o fato observado correspondente é gerado e inserido na descrição de fatos.

A descrição dos fatos observados, em conjunto com a descrição da biblioteca de planos, são utilizadas pelo transformador *FindPlan* que reconstrói o conjunto de planos possíveis de serem utilizados na explicação dos fatos observados. Este transformador é apresentado nas figuras 53 e 54. Seu

funcionamento é guiado pela transformação *mainTransform* do conjunto *MainSet*. Esta transformação reconhece o programa de descrição dos fatos observados, identifica a biblioteca de planos (*aLibPlan*) a ser utilizada e a lista de fatos observados (*ip*) e aplica os conjuntos de transformadores que tratam cada um destes elementos: *HandlePlanLibrary* que lê a biblioteca de planos, extrai informações sobre os eventos da biblioteca e insere estas informações em uma base de conhecimentos[*]; *HandleObservedFacts* que, a partir de cada fato observado, gera a lista de eventos inferidos; e finalmente o conjunto *CalculateEndEventValue* que calcula o valor de cada evento final inferido no processo. O valor do evento final corresponde ao seu grau de satisfação e indica o quanto este evento teve suas regras atendidas a partir dos fatos observados.

O resultado da aplicação da técnica de reconhecimento de planos é formado por vários conjuntos de eventos finais, onde cada conjunto explica todos os fatos observados. O conjunto a ser escolhido como resultado final pode ser aquele que possuir o maior grau médio de satisfação dos eventos ou aquele que possuir o menor número de eventos finais.

[*] A Máquina Draco-Puc possui uma base de conhecimento (estilo Prolog) implementada para permitir o compartilhamento de informações entre transformações.

Transformer FindPlan (1/2)			
	Set of Transforms MainSet		
	Transform MainTransform		
		LHS	loadLibrary [[String aLibName]] . [[instanceProgram ip]]
		Pre-Apply	*1) ler o módulo aLibName e colocar em definitionsLoc* *2) Aplicar o cj de transformações HandlePlanLibrary em definitionsLoc* *3) Aplicar o cj de transformações HandleObservedFactsSet em ip* *4) Aplicar o cj de transformações CalculateEndEventValue em WSFinal*
	End	*Salvar WSFinal*	
External	**Set of Transforms HandlePlanLibrary**		
	Transform LocateEvent		
		LHS	Event [[Name en]] [[parListdecl al]] [[eventAttribute ea]] [[declBody db]]
		Pre-Apply	*1) Armazena as informações sobre o evento na KB* *- evento final ou não / número de parâmetros da regra*
	Transform putDecompositionValuesInKB		
		LHS	composedBy [[decompositionStatement* ds]]
		Pre-Apply	*1) Obtém uma lista com a ordem dos parâmetros do evento todo* *2) Enquanto existir evento parte em ds* *- aplicar o template FindNextPartOfDeclaration em ds*
	Template FindNextPartOfDeclaration		
	External	**LHS**	[[Name en]] [[parListdecl ald]] by [[Name rn]] [[parameters p]] ;
		Pre-Apply	*1) Obtém o valor do parâmetro weight em p* *2) Para cada parâmetro em ald acha a ordem correspondente no evento todo* *3) Insere as informações na KB*
	Set of Transforms HandleObservedFactsSet		
	Transform LocateObservedEvent		
		LHS	[[Name anEvent]] [[argList pl]] [[value v]]
		Pre-Apply	*1) Faz currentObservedEventName ser igual ao evento atual* *2) Copia a justificativa do evento para rationaleForCurrentInferredEvent* *3) Aplicar o cj de transformações FindPlanSet em definitionsLoc* *4) Enquanto existir evento que foi inferido* *a) Mover os eventos inferidos para WSCurrentObservedEvents* *b) Aplicar o cj de transformações HandleGeneratedFactsSet em WSCurrentObservedEvents* *5) Aplicar o cj de transformações MergeEndEvents em WSEndEvents*
	Set of Transforms FindPlanSet		
	Transform LocateCurrentEventDeclaration		
		LHS	Event [[Name en]] [[parListdecl ad]] [[eventAttribute ea]] [[declBody db]]
		Pre-Apply	*1) SE currentObservedEventName for igual a en* *Aplicar o cj de transformações HandleSpecializationRule em db* *SENÃO Aplicar o cj de transformações HandleDecompositionRules em db*
	Set of Transforms HandleDecompositionRules		
	Transform CurrentEventDeclaredAsPartOf		
		LHS	[[Name en]] [[parListdecl al]] by [[Name rn]] [[parameters p]] ;
		Pre-Apply	*1) SE currentObservedEventName for igual a en* *Aplicar o cj de transformações AddNewDecompositionInferredEvent em WSEndEvents*
	Set of Transforms HandleSpecializationRule		
	Transform GeneralizationForCurrentEvent		
		LHS	isa [[Name n]]
		Pre-Apply	*1) Aplicar o cj de transformações AddNewGeneralizationInferredEvent em WSEndEvents*

Figura 54 – Transformador de Reconhecimento de Planos

Transformer FindPlan (2/2)		
External	**Set of Transforms AddNewGeneralizationInferredEvent**	
	Transform AddNewGeneralizationInferredEventTransformation	
	LHS	Recognition begin [[topInferredEvent* oe]] end
	Pre-Apply	*1) Copia parâmetros de rationaleForCurrentInferredEvent para pl* *2) SE evento inferido já foi inferido anteriormente ENTÃO SKIP_APPLY* *3) SE evento inferido é um evento final ENTÃO* *a) Adicionar o evento inferido em WSEndEvents* *b) SKIP_APPLY*
	RHS	Recognition begin [[topInferredEvent* oe]] [[Name en]] [[argList pl]] { isa [[anyEvent rat]] } . end
	Set of Transforms AddNewGeneralizationInferredEvent	
	Transform AddNewGeneralizationInferredEventTransformation	
	LHS	Recognition begin [[topInferredEvent* oe]] end
	Pre-Apply	*1) Obtém os parâmetros de rationaleForCurrentInferredEvent para pl* *2) Cria os parâmetros do novo evento inferido levando em consideração:* *- a diferença de versões entre os parâmetros do novo evento inferido (todo) e da parte* *- os parâmetros coringas ("*") do novo evento inferido* *2) SE evento inferido já foi inferido anteriormente ENTÃO SKIP_APPLY* *3) SE evento inferido é um evento final ENTÃO* *a) Adicionar o evento inferido em WSEndEvents* *b) SKIP_APPLY*
	RHS	Recognition begin [[topInferredEvent* oe]] [[Name en]] [[argList pl]] { [[Name ruleName]] : [[anyEvent rat]] } . end
	Set of Transforms HandleGeneratedFactsSet	
	Transform LocateGeneratedEvent	
	LHS	[[Name anEvent]] [[argList pl]] [[value aValue]] { [[rationale rat]] } .
	Pre-Apply	*1) Faz currentObservedEventName ser igual ao evento atual* *2) Copia a justificativa do evento para rationaleForCurrentInferredEvent* *3) Aplicar o cj de transformações FindPlanSet em definitionsLoc*
	Set of Transforms MergeEndEvents	
	Transform LocateNewEndEvent	
	LHS	[[Name aName]] [[argList pl]] [[value aValue]] { [[rationaleStatement* rat]] } .
	Pre-Apply	*1) PARA cada evento em WSFInal igual ao evento atual* *a) SE listas de parâmetros equivalentes ENTÃO* *i) SE for necessário usar coringas nos parâmetros ENTÃO integrar os parâmetros e duplicar os eventos corrente e existente* *i) Encontrar o primeiro ponto diferente entre o evento atual e o existente* *ii) Modificar o evento existente adicionando o evento atual a partir do ponto encontrado* *2) SE o evento corrente não tiver sido processado ENTAO inserir em WSFinal*
	Set of Transforms CalculateEndEventValue	
	Transform LocateNewEndEvent	
	LHS	[[Name aName]] [[argList pl]] [[value aValue]] { [[rationaleStatement* rat]] }.
	Pre-Apply	*1) Percorre rat calculando aNewValue*
	RHS	[[Name aName]] [[argList pl]] [[value aNewValue]] { [[rationaleStatement* rat]] }.

Figura 55 – Transformador de Reconhecimento de Planos (continuação)

```
PlanLibrary
begin
  Event f1(A)
    begin
      isa i1
    end
  Event f2(A);
  Event i1(A)
    begin
      isa i3
    end
  Event i2(A)
    begin
    isa E1
    composedBy
      f2(A) by r1 with weight = '1.0';
      i3(A) by r2 with weight = '1.0';
    end
  Event i3(A)
    begin
      isa E2
    end
  Event E1(A)  is EndEvent ;
  Event E2(A)  is EndEvent
  begin
      isa i3
  end
end
```

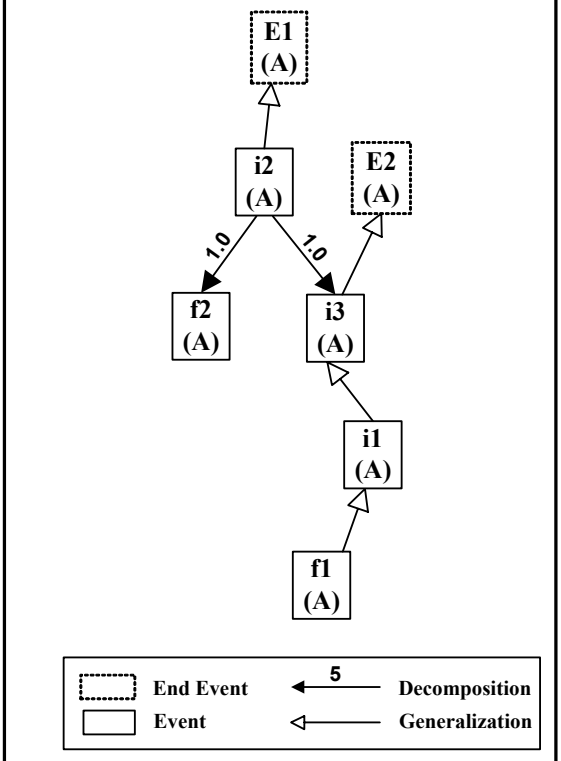

Figura 56 – Exemplo de biblioteca de planos

Para exemplificar a reconstrução dos planos a partir dos fatos observados será utilizado o exemplo da biblioteca de planos da figura 56, que utiliza um pequeno caso genérico para facilitar o entendimento e permitir que sejam mostrados os tratamentos das diversas construções possíveis. A seqüência de geração dos eventos inferidos pelo transformador *FindPlan*, para o caso em que foram observados os fatos *f1* e *f2*, é mostrada na figura 57. O transformador utiliza 3 áreas de trabalho auxiliares: *WSCurrentObservedFacts* que contém os eventos inferidos e que ainda necessitam serem tratados, ou seja, não são eventos finais; *WSEndEvents* que contém o conjunto de eventos finais inferidos a partir de um único evento observado; e *WSFinal* que armazena o conjunto de eventos finais que explicam os eventos observados que já foram tratados.

O transformador trata um evento observado por vez, primeiramente inserindo os eventos diretamente inferidos a partir do evento observado em *WSCurrentObservedEvents* (figura 57: interação 1 para *f1* e 6 para *f2*). Em seguida, enquanto houver algum evento em *WSCurrentObservedEvents*, este evento é tratado de maneira a gerar os novos eventos inferidos a partir dele (figura 57: interações 2 a 4 para *f1* e 7 para *f2*). Estes eventos são inseridos em *WSCurrentObservedEvents*, se não forem eventos finais, ou em *WSEndEvents*

caso contrário. Após todos os eventos possíveis de serem inferidos tiverem sido identificados e tratados é realizada a integração de cada evento final de *WSEndEvents* com os eventos finais pré-existentes em *WSFinal* (figura 57: interação 5 para *f1* e 8 para *f2*). Esta integração é realizada a partir do ponto comum mais alto da hierarquia de eventos .

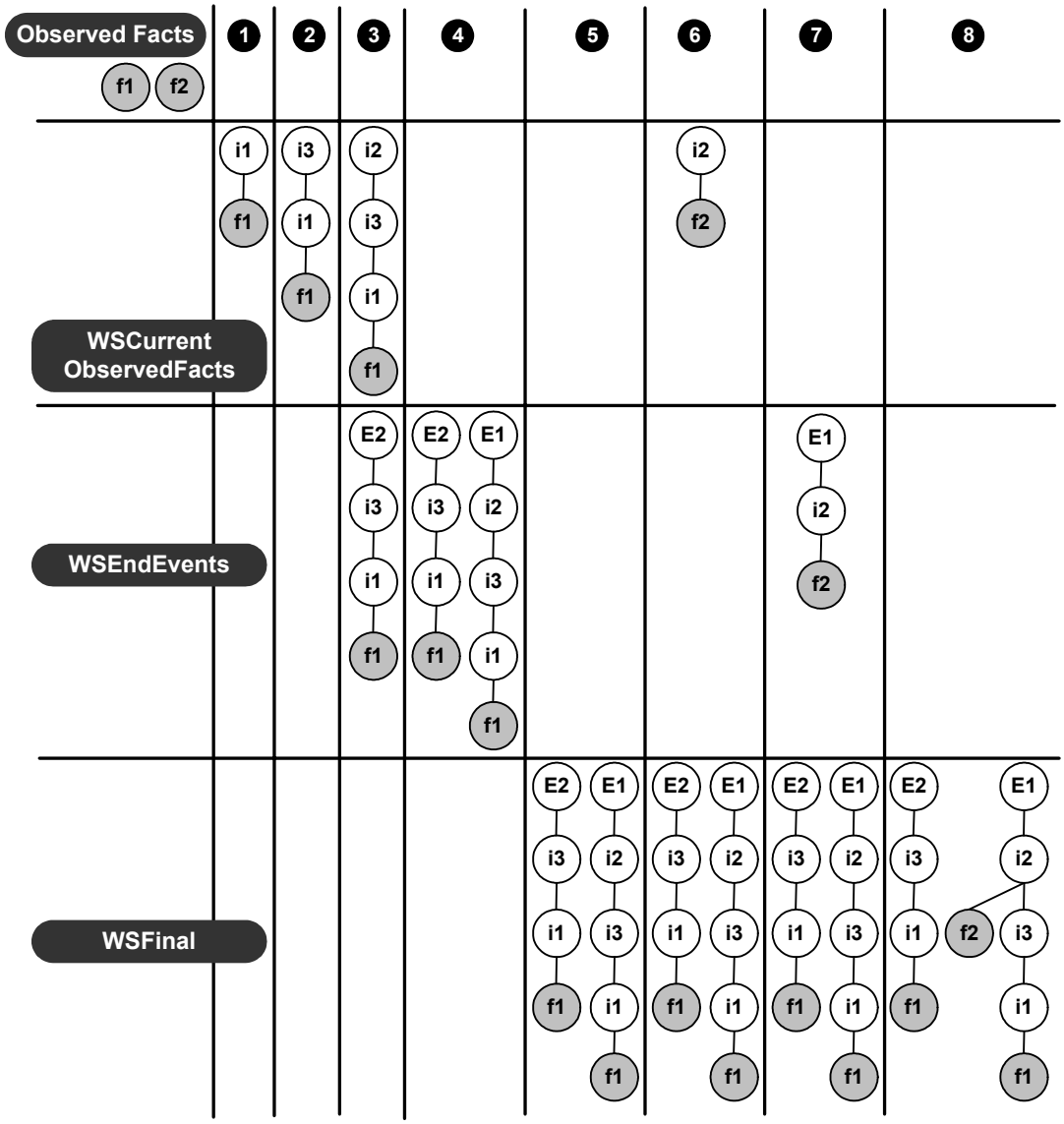

Figura 57 – Visualização do funcionamento do transformador de reconhecimento de planos

O resultado final da aplicação do transformador *FindPlan*, descrito utilizando-se o elemento *planRecognition* da linguagem apresentada na figura 51, é mostrado a seguir. Cada evento final identificado possui um valor associado (seu grau de satisfação), calculado a partir do valor do evento observado e dos pesos das regras de decomposição especificados na biblioteca de planos:

Recognition
begin

```
  E2 ('obs') [' 1.00'] {
    isa i3 ('obs')  {
      isa i1 ('obs')  {
        isa f1('obs')['1']
      }
    }
  }.
  E1 ('obs') [' 1.00'] {
    isa i2 ('obs')  {
      r2:i3 ('obs')  {
        isa i1 ('obs')  {
          isa f1('obs')['1']
        }
      }
      r1:f2 ('obs')['1']
    }
  }.
End
```

4.3. Refinamento de Modelos

A atividade de refinamento do modelo permite a aplicação das transformações anteriormente geradas de maneira a evoluir automaticamente um artefato de uma versão para a versão seguinte. Esta atividade permite que o gerenciador de configuração armazene apenas uma versão inicial de cada modelo e as transformações que produzem cada nova versão.

A execução desta atividade é mostrada na figura 58. Inicialmente o desenvolvedor descreve suas intenções de refinamento em um arquivo de *script*. Esta descrição especifica qual a versão corrente a ser refinada, qual o nome da próxima versão e o nome do transformador correspondente.

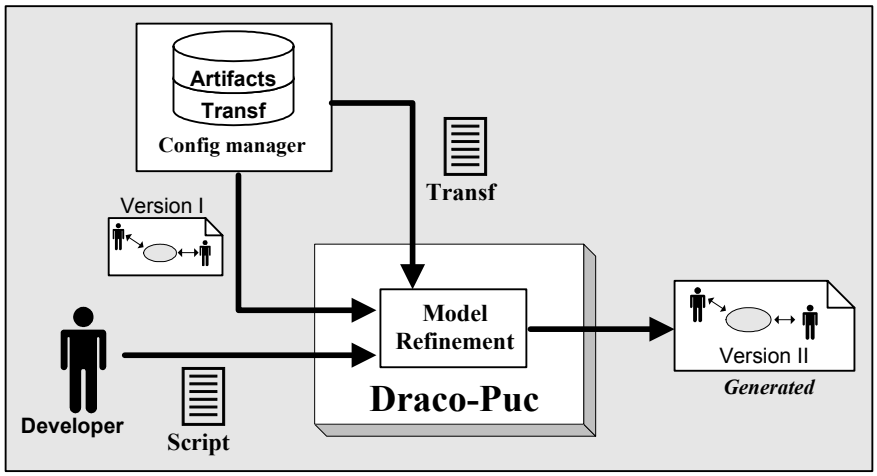

Figura 58 – Processo de Refinamento de Modelos

O refinamento propriamente dito é realizado através da aplicação do transformador *evolveArtifact* sobre o arquivo de *script*. Este transformador é

mostrado na figura 59. Cada especificação de refinamento do *script* é identificada pela transformação *FindEvolveStatement* que inicialmente lê o artefato a ser refinado (*aSource*) e o transformador de rastreamento correspondente (*aSOTName*). Este transformador é aquele gerado na atividade anterior e se encontra armazenado como DAST, evitando que seja necessário executar qualquer passo intermediário de geração de código ou compilação. Em seguida este transformador de rastreamento é aplicado sobre o artefato original (linha 3) gerando a nova versão do artefato. Finalmente o artefato modificado é salvo com o nome definido para a nova versão pelo arquivo de script (*aTarget*).

Transformer EvolveArtifact				
	Set of Transforms MainSet			
		Transform FindEvolveStatement		
			LHS	evolve [[String aSource]] to [[String aTarget]] using [[String aSOTName]] ;
			Pre-Apply	*1) ler o módulo aSource* *2) ler o módulo aSOTName e colocar em aNewSOT* *3) Aplicar o cj de transformações aNewSOT em aSource* *4) Salvar aSource como aTarget*
External	**Set of Transforms aNewSOT**			
		Transform		
			LHS	
			RHS	

Figura 59 – Transformador de Refinamento

A figura 60 mostra um exemplo da execução deste processo. A transformação *FindEvolveStatement* identifica o artefato original (*cen1_1.xml*), o nome do novo artefato (*cen1_2.xml*) e o transformador de rastreamento (*SOTcen1_1-cen1_2.tlb*). O arquivo *SOTcen1_1-cen1_2.tlb* contém a forma interna utilizada pela Máquina Draco-PUC do transformador de rastreamento gerado na atividade anterior, que pode ser diretamente lida e aplicada sobre o artefato original para gerar o novo artefato.

Esta atividade pode ainda ser utilizada no processo de desenvolvimento de um novo artefato em um contexto diferente daquele em que o transformador de rastreamento foi gerado, ou seja, utilizar transformadores anteriormente gerados para outros artefatos no refinamento de um novo artefato. Neste caso o desenvolvedor especifica o contexto do seu trabalho, seleciona um plano dentre um conjunto de planos aplicáveis à situação e que foram identificados pelo sistema, e finalmente o sistema aplica as transformações ao artefato em desenvolvimento, conseguindo desta maneira a reutilização do plano e das transformações geradas em um novo contexto.

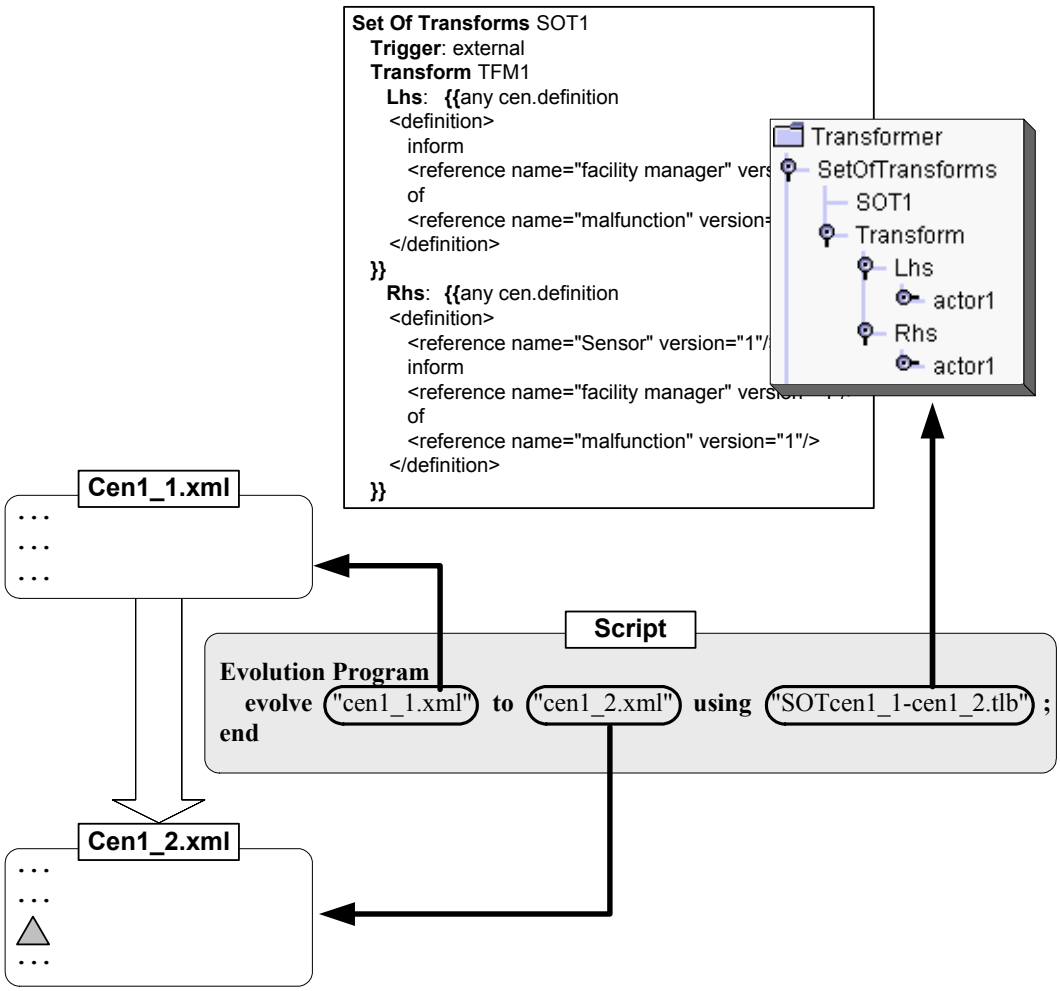

Figura 60 – Exemplo de refinamento de modelo

4.4. Conclusão

Neste capítulo foi apresentada uma proposta de mecanismo de rastreamento baseado em transformações. Este mecanismo permite a captura automática das informações sobre a evolução dos diversos artefatos através da identificação das mudanças existentes entre duas versões consecutivas de um artefato. Estas informações constituem as ligações Evolui-Para e Justifica do modelo de referência proposto por Ramesh & Jarke [3].

Através da utilização do mecanismo proposto são capturadas e armazenadas informações de rastreamento na forma: de transformações a serem aplicadas em uma versão de um artefato de maneira a gerar sua versão seguinte; planos que explicam o conjunto de diferenças identificadas entre as versões; e informações sobre o contexto das mudanças realizadas. Em função da disponibilidade destas

informações e do processo utilizado na sua captura será apresentada na seção seguinte uma análise do mecanismo de rastreamento proposto neste capítulo frente aos requisitos necessários aos sistemas de rastreamento previamente apresentados na seção 3.3.2.

4.4.1. Análise da Satisfação dos Requisitos

Requisito: Representação da semântica e dos atributos relacionados a cada ligação
Satisfação do Requisito: SIM
Justificativa: A representação da semântica e dos atributos das ligações é realizada diretamente através do resultado das ações realizadas pelo desenvolvedor. Desta maneira qualquer atributo necessário pode ser diretamente calculado a partir das informações. Por exemplo, a importância de uma ligação Evolui-Para pode ser diretamente calculada em função de qual o plano que foi reconhecido, do contexto e das ações necessárias a obter a nova versão.

Requisito: Variação da Granuralidade das tarefas de rastreamento de acordo com a importância e a criticidade dos artefatos relacionados
Satisfação dos Requisitos: SIM
Justificativa: Dois aspectos podem ser utilizados pelo mecanismo para variar a granularidade das tarefas de rastreamento. O primeiro envolve a definição da quantidade de ações realizadas pelo desenvolvedor entre o momento do *check-out* e do *check-in*. A gerência do projeto pode variar o intervalo entre os diversos acionamentos do mecanismo, em função da criticidade e da importância dos artefatos. O segundo aspecto relaciona-se a própria construção da biblioteca de planos. O mecanismo permite que se definam planos em diversos níveis de abstração, ou seja, os planos da biblioteca a serem reconhecidos podem ser representativos de ações atômicas ou de um conjunto de ações. A escolha de qual biblioteca utilizar, a mais ou a menos abstrata, pode, também, ser definida em termos da criticidade e da importância dos artefatos.

Requisito: Disponibilizar serviços de inferência com suporte a semântica dos diversos tipos de ligações
Satisfação dos Requisitos: SIM
Justificativa: A Máquina Draco-PUC pode ser utilizada para gerar qualquer nova informação a partir das transformações e planos existentes.

Requisito: Permitir o uso tanto de representações formais quanto de representações informais das informações de rastreamento
Satisfação dos Requisitos: SIM
Justificativa: As informações de rastreamento capturadas e armazenadas pelo mecanismo são compostas pelas transformações de rastreamento e pelos planos que foram reconhecidos como possíveis de serem utilizados para explicar estas ações. Podem ser criados conjuntos de transformações que, aplicadas sobre as informações de rastreamento capturadas, gerariam as representações formais ou informais correspondentes. A construção destas transformações é uma tarefa

simples e a sintaxe das representações geradas é totalmente flexível, ou seja, não está amarrada a nenhuma sintaxe fixa.

Requisito: Permitir a definição de versões, o gerenciamento da configuração do sistema e possuir um mecanismo de notificação de mudanças.
Satisfação dos Requisitos: Pode ser implementado
Justificativa: O mecanismo utiliza os conceitos de versões e configurações, mas não implementa os mecanismos necessários a sua definição e gerenciamento. Na seção 5.2 é mostrado um exemplo de utilização destes recursos no contexto do protótipo do sistema de rastreamento. A melhor alternativa para disponibilizar o conjunto completo destas funcionalidades é através da utilização de um sistema de controle de versões existentes, de código aberto e nos moldes do CVS, integrado ao mecanismo proposto. Esta alternativa é bastante plausível de ser implementada, devido ao fato de que, tanto as transformações de rastreamento quanto os planos reconhecidos, poderem ser tratados como arquivos comuns de qualquer sistema operacional.

Requisito: Integração com o Processo de Desenvolvimento
Satisfação dos Requisitos: SIM
Justificativa: O processo de captura das informações de rastreamento pode ser totalmente integrado não só ao processo de desenvolvimento, mas também a qualquer ferramenta utilizada pelo desenvolvedor e que possibilite a exportação dos modelos no formato XML. A utilização do mecanismo em outros tipos de artefatos é apresentada na seção 5.4 e envolve, principalmente, a definição de novas bibliotecas de planos.