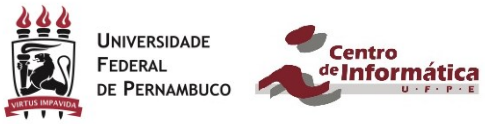
UNIVERSIDADE FEDERAL DE PERNAMBUCO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
CENTRO DE INFORMÁTICA



CRITÉRIOS DE COBERTURA DE TESTES GERADOS A PARTIR DE LINGUAGEM NATURAL

TRABALHO DE GRADUAÇÃO

Aluno: Tomaz de Aquino dos Santos Junior (tasj@cin.ufpe.br)
Orientador: Augusto Sampaio (acas@cin.ufpe.br)
Co-orientador: Gustavo Carvalho (ghpc@cin.ufpe.br)

RECIFE
2015

TOMAZ DE AQUINO DOS SANTOS JUNIOR

CRITÉRIOS DE COBERTURA DE TESTES GERADOS A PARTIR DE LINGUAGEM NATURAL

Trabalho de conclusão do curso de
Bacharelado de Ciências da Computação da
Universidade Federal de Pernambuco,
realizado sob a orientação dos Professores
Augusto Sampaio e Gustavo Carvalho

RECIFE
2015

Agradecimentos

A Deus por permitir a presença dos meus pais em vida, após todos os acontecimentos do período de desenvolvimento deste trabalho, câncer de minha mãe, juntamente com a cirurgia e quimioterapia, AVC do meu pai, que se recuperou com grande força de vontade.

A minha família, os esforços dos meus pais em dispor a melhor educação possível, incentivos, conselhos, junto dos grandes exemplos morais e espirituais que são em minha vida.

A minha nova família, minha esposa Jéssika e meu filho Miguel, atualmente com 7 meses de idade. Ela sempre me dando força nos momentos mais difíceis, segurando a barra quando precisei. Ele sendo a luz que tem iluminado a minha vida.

A meus amigos de faculdade: Marcos, Filipe, Paulo, Bruna, João Paulo e Jonatas, que me proporcionaram momentos maravilhosos dentro do Centro de Informática, novos conhecimentos técnicos e de vida.

A meus amigos fora da faculdade: Pedro, Jonatas, Naevio, Rangel que sempre se preocuparam comigo nos acontecimentos destes últimos meses de minha vida.

Aos meus orientadores Augusto e Gustavo e seu aluno de Mestrado, Hugo, pela ajuda e compreensão durante todo o processo.

"Embora ninguém possa voltar atrás e fazer um novo começo, qualquer um pode começar agora e fazer um novo fim."

Chico Xavier

Resumo

A complexidade de software e hardware produzidos tem aumentado rapidamente nos últimos anos, o que resulta em um maior esforço para garantir a corretude destes, fator importante no desenvolvimento de qualquer sistema; em particular, daqueles tidos como críticos.

Consequentemente, é essencial verificar o correto funcionamento de tais sistemas. Uma das estratégias é o uso de métodos de geração automática de testes, o que tem levado ao surgimento de ferramentas baseadas em diferentes modelos matemáticos. Uma dessas ferramentas é a NAT2TEST, desenvolvida no contexto de uma parceria entre o Centro de Informática da UFPE e a Embraer, que possui o objetivo de gerar casos de teste a partir de requisitos descritos em uma Linguagem Natural Controlada (um subconjunto do Inglês com uma gramática precisamente definida). Apesar de gerar testes, a NAT2TEST não implementa qualquer critério de cobertura.

O objetivo deste trabalho é criar critérios de cobertura para os testes gerados a partir da NAT2TEST, através do controle da ferramenta FDR, que é um verificador de modelos para na linguagem CSP, utilizando a linguagem de script TCL.

Palavras-Chave: geração automática de teste, CSP, critérios de cobertura

Sumário

Lista de Figuras

Lista de Tabelas

1. Introdução

Ao longo dos últimos anos, o crescimento de componentes de software e hardware em Sistemas Críticos tem aumentado significativamente. Um relatório feito pela NASA [11] mostra que de 1960 a 2000, a quantidade de software embarcado em artefatos militares aumentou de 8% para 80%; crescendo a necessidade de métodos que possam garantir a corretude dos mesmos.

A área de Testes de Software surge com o objetivo de melhorar a qualidade do software revelando seus erros, porém não garante a total ausência destes, pelo fato dos programas atuais possuírem grande complexidade, resultando em uma quantidade potencialmente infinita de possibilidades a se testar [4]. Ferramentas de verificação de modelos (*model checking*) têm o objetivo de efetuar uma análise exaustiva, garantido aderência a certas propriedades de interesse, mas, na prática, possuem o conhecido problema de explosão de estados, impedindo a análise completa de sistemas complexos reais. Apesar de não garantirem a ausência total de erros, testes continuam sendo a alternativa de verificação mais utilizada na prática.

Existem várias ferramentas que auxiliam o processo de criação de casos de testes de forma manual ou automática. As ferramentas de elaboração automática estão menos propensas ao erro humano e possuem a capacidade de gerar testes de forma exaustiva ou seguindo algum critério de cobertura em especial. Neste trabalho, considera-se a ferramenta NAT2TEST [1].

A NAT2TEST é uma aplicação prática da abordagem *MBT* (Model-Based Testing) [5] cuja principal funcionalidade é a geração automática de

casos de teste a partir de modelos que descrevem o comportamento esperado do sistema. Neste trabalho, o modelo é automaticamente gerado a partir de requisitos escritos em linguagem natural controlada. A ferramenta foi desenvolvida no contexto de uma parceria entre o Centro de Informática da UFPE e a Embraer.

1.1. Objetivos

A ferramenta NAT2TEST é baseada em uma linguagem natural controlada [2] que possibilita a especificação de requisitos temporais. Estes requisitos descrevem o comportamento esperado do sistema para cenários específicos. A NAT2TEST representa formalmente este comportamento utilizando o modelo DFRS (*Data-Flow Reactive System*) [12]. Este modelo, por sua vez, pode ser traduzido para a linguagem CSP (*Communicating Sequential Process*) [6], entre outras alternativas, com o intuito de gerar testes via contraexemplos de verificação de refinamentos, usando a ferramenta FDR [7].

O objetivo desse trabalho é estender a ferramenta NAT2TEST para que esta possa gerar testes considerando diferentes critérios de cobertura. Com essa nova característica, é possível criar casos de testes mais direcionados aos objetivos da campanha de testes, por exemplo, ter pelo menos um caso de teste para cada requisito do sistema.

1.2. Estrutura do Documento

Capítulo 2: Esse capítulo descreve conceitos relacionados à geração automática de testes, que servem como base para o entendimento deste trabalho.

Capítulo 3: Contém uma visão geral do funcionamento da ferramenta NAT2TEST, além de explicar os critérios de cobertura considerados e os passos necessários para a implementação dos mesmos.

Capítulo 4: Conclui o documento, com um resumo das contribuições do trabalho e possíveis trabalhos futuros relacionados ao projeto.

2. Conceitos da Geração Automática de Testes

Para o entendimento deste trabalho são necessários alguns conceitos gerais relacionados às técnicas e notações utilizadas. A ferramenta NAT2TEST [1] é mais elaborada do que pode ser capturado da visão geral mostrada a seguir, no entanto, o objetivo deste capítulo é situar o leitor dentro do escopo de geração de testes. Mais informações sobre a ferramenta NAT2TEST são apresentados no Capítulo 3.

De forma resumida, a NAT2TEST utiliza MBT [4] como abordagem para gerar os casos de testes através da linguagem CSP, método formal usado para representar o comportamento esperado do sistema. Após a geração do CSP, utiliza-se a ferramenta FDR para gerar testes através da verificação de refinamentos.

2.1. Model-Based Testing

MBT (*Model-Based Testing*) é uma técnica de Geração Automática de Testes utilizando modelos dos requisitos e comportamento do sistema. Apesar deste tipo de geração automática requerer um esforço significativo inicial na construção do modelo, ele oferece vantagens em relação a outras técnicas [4].

Geralmente, o modelo é criado manualmente a partir das especificações ou requisitos do sistema. No caso da NAT2TEST, o modelo, chamado de DFRS (*Data-Flow Reactive System*), é gerado automaticamente com o uso de algoritmos que processam os requisitos descritos em uma Linguagem Natural Controlada (subconjunto do Inglês com uma gramática precisamente definida). Em seguida, este modelo é traduzido para uma outra notação, sendo uma das alternativas a álgebra de processos CSP

(*Communicating Sequential Processes,* mais detalhado posteriormente) [3] que descreve o comportamento do sistema.

Após a criação do modelo, testes são gerados automaticamente, através de alguma ferramenta. No caso da NAT2TEST, a partir do modelo em CSP, utilizam-se as ferramentas FDR [7] e Z3 [10].

Uma falha em algum dos testes indica que o comportamento do sistema não coincide com o comportamento descrito no modelo, o que resulta geralmente de uma implementação que não está em conformidade, ou de erro na própria modelagem [4].

2.1.1. Vantagens e Desvantagens

Como dito anteriormente, a abordagem MBT possui vantagens em comparação com a geração manual de casos de teste. Por exemplo:

- Maior facilidade de manutenção, pois não é necessário escrever novos testes para cada novo recurso;
- Os modelos são independentes da implementação do sistema e podem ser feitos ao mesmo tempo por equipes diferentes.

Entretanto, a abordagem MBT também possui desvantagens, como:

- Obrigatoriedade da representação do sistema através de um modelo
- Qualidade do modelo influencia a qualidade dos testes gerados
- Os modelos gerados podem ser muito complexos, levando a uma dificuldade na geração de testes

2.2. Métodos Formais

Métodos Formais são técnicas matemáticas utilizadas na especificação, desenvolvimento e verificação de software [8]. O seu uso em projetos de sistemas é motivado pelo fato do cunho matemático permitir uma maior consistência e análise aprofundada, através da verificação de propriedades.

Os métodos formais podem ser utilizados em diferentes fases de um projeto de software. Na fase de especificação, para descrever o funcionamento do sistema a ser desenvolvido, no desenvolvimento, para verificar se o comportamento da parte desenvolvida está em conformidade com a especificação descrita, e após o desenvolvimento, para verificar propriedades da implementação final.

No escopo da ferramenta NAT2TEST utiliza-se a linguagem formal CSP para descrever o comportamento dos sistemas a serem analisados, e a ferramenta FDR para fazer verificações utilizadas na geração de contraexemplos. Aspectos temporais da modelagem são descritos de forma simbólica e a geração de valores concretos é realizada pela ferramenta Z3. Como a parte temporal não está no escopo deste trabalho, não detalhamos as funcionalidades da ferramenta Z3.

2.2.1. CSP

CSP é uma linguagem formal utilizada para descrever a interação entre sistemas concorrentes, e tem sido muito utilizada como uma ferramenta de especificação e verificação de propriedades de sistemas [3].

Possui como estrutura básica um Processo, entidade autônoma que possui interfaces, que são como portas de entradas ou saídas para interação

com o ambiente através de eventos. Dois ou mais processos combinados podem formar outro processo [3].

2.2.1.1. Processos

Um processo representa uma unidade de comportamento, podendo ser utilizada para a formação de outros processos. É composto por eventos, sendo estes indivisíveis e instantâneos, representados por nomes simples ou compostos, podendo ser também de entrada ou saída [3].

O conjunto de eventos de um processo é conhecido como o alfabeto do processo. Estes eventos representam ações ou estados e possuem identificadores únicos. Uma breve introdução será apresentada a seguir, utilizando a notação CSP_M [6], versão do CSP legível por máquina.

Um exemplo de processo básico em modelos CSP é o STOP. O processo STOP, é um exemplo de processo primitivo, pois não possui nenhum evento em seu alfabeto e não se comunica com nenhum outro processo. Sua função é a de informar um *deadlock*. Por exemplo, considere o seguinte processo:

```
P = a -> b -> STOP
```

Esse exemplo apresenta um trecho de código CSP_M que declara um processo `P`, contendo os eventos `a` e `b` em seu alfabeto. O comportamento de `P` é esperar indefinidamente até que o evento `a` aconteça, após este, espera até que o evento `b` aconteça, quando se comporta como `STOP`. Em resumo, isso significa que ele comunica `a`, depois `b` e em seguida tem o comportamento de um *deadlock* [3].

Outro processo básico em CSP é o SKIP. Este processo indica terminação com sucesso e é definido pelo processo `_tick -> STOP`. Além

do operador de prefixo (->), outros operadores que serão de constante uso neste trabalho estão definidos abaixo.

2.2.1.2. Operadores de CSP

- Composição sequencial (`;`):

O operador de composição sequencial executa o processo do lado esquerdo até que ele termine com sucesso. Em seguida, executa o processo do lado direito. Por exemplo, o processo `P;Q` executa o processo `P` até que ele termine e em seguida excuta o processo `Q`.

- Escolha externa (`[]`)

Esse operador tem a função de deixar que o ambiente escolha qual evento entre os processos envolvidos será executado. Por exemplo, no processo `P[]Q`, se o primeiro evento escolhido pertencer somente à `P`, então o processo como um todo se comporta como `P`. Caso um evento de `Q` seja escolhido, o processo se comporta como Q.

- Paralelismo generalizado (`[|{X}|]`)

Sincroniza eventos que pertençam ao conjunto `X`. Cada processo se desenvolve independentemente em eventos que não pertençam à `X`, porém essa composição só termina com sucesso quando ambos os lados terminam.

- Interleaving (`|||`)

Esse é outro operador de paralelismo, porém considera que ambos os processos podem comunicar os eventos livremente, sem sincronização. O processo `P|||Q` é equivalente ao processo `P[|{}|]Q`.

2.2.1.3. Outras Características

Além de eventos, os processos podem fazer uso de tipos de dados. A notação CSP oferece suporte a dados primitivos como inteiros e booleanos e também a liberdade de construir novos tipos [3].

CSP ainda permite o uso de canais de comunicação para fazer transmissão de dados, incluindo *inputs* e *outputs*. Esses canais são tipos especiais de eventos, porém com a função de comunicação de certo tipo de dado. Por exemplo, o código `channel in:Integer` declara um canal `in` que transmite dados do tipo inteiro.

2.2.1.4. Traces

No nível de abstração provido por CSP, processos interagem com o ambiente através da atuação de eventos em sua interface. O ambiente, sendo um processo, um usuário, ou a mescla destes, não tem acesso direto ao estado interno do processo ou aos eventos que este realiza [3].

Para analisar o comportamento de processos, é necessário considerar a sequência de eventos que podem ser observados na interface do mesmo. Essas observações são chamadas *traces*. O conjunto de todos os possíveis *traces* de um processo `P` é denominado `traces(P)` [3].

Em resumo, um *trace* é o registro dos eventos na ordem que eles ocorrem na execução de um processo [3]. O conjunto de todos *traces* denota todas as possíveis sequências de eventos em todas as possíveis execuções de um processo.

```
traces(SKIP)={<>,<_tick>}

traces(STOP)={<>}

P = a → b → STOP

traces(P) = {<>,<a>,<a,b>}
```

O processo SKIP possui apenas o evento `_tick`, logo pode executar nenhum evento ou apenas `_tick`. O processo STOP não possui eventos. O processo P possui os eventos `a` e `b`, porém para o evento `b` ocorrer, o evento `a` já deve ter ocorrido, portanto as possíveis execuções são `<>, <a>, <a,b>`.

CSP possui outros modelos mais elaborados (como o modelo de falhas, que permite análise de *deadlock*, e o de falhas-divergências, que permite análise de *livelock*). Estes modelos são brevemente discutidos na Seção 2.3 a seguir.

2.2.2. Verificação de Propriedades

Por ser uma notação matemática, a álgebra de processos CSP permite a verificação de propriedades dos processos descritos, o que é bastante útil no estudo do comportamento de sistemas [3]. A ferramenta utilizada para tais verificações chama-se FDR (Failures-divergences refinement) [7], vista com detalhes a seguir.

2.3. FDR

Verificadores de modelos possuem grande importância no processo de geração automática de testes. Através deles é possível obter informações (explicadas neste capítulo) acerca do modelo utilizado para representar o comportamento do sistema, no caso do NAT2TEST, a notação CSP.

A ferramenta FDR foi desenvolvida pela *Formal Systems* e é utilizada para verificação de modelos CSP. Os casos de teste são obtidos pela NAT2TEST a partir de contraexemplos resultantes de refinamentos mal sucedidos feitos pelo FDR entre o CSP original que representa o comportamento do sistema e outro modelo CSP modificado com uma "marca".

Esta marca possui o objetivo de controlar os contraexemplos gerados, do qual são extraídos os testes. Todo este processo será visto com mais detalhes no capítulo 4.

A NAT2TEST faz chamadas ao FDR através de linhas de comando no terminal, recebendo o arquivo CSP como parâmetro. Para isto, utiliza uma API implementada por Freitas e Woodcock [9], que possibilita manipular o FDR através de scripts escritos na linguagem de programação TCL.

O FDR é capaz de converter os processos CSP em modelos LTS [5] (*Labelled Transition Systems*, conceito usado para representar o comportamento de sistemas discretos, através de estados e transições) e então determinar se um dos processos é refinado pelo outro, considerando um dos possíveis modelos semânticos: *traces*, falhas, falhas/divergências [7].

- Refinamento por traces: Este modelo é baseado na sequência de eventos que um processo executa (*traces* de um processo). O processo P é um refinamento por *traces* de Q, se todas as possíveis

sequencias de P forem possíveis em Q. Se o refinamento não ocorre, é retornado o contraexemplo mais curto.

- Refinamento por falhas: A falha é um par `(s,X)` onde `s` é um trace do processo e `X` é um conjunto de eventos que o processo pode recusar executar. P refina Q por falhas se o conjunto das falhas de Q está incluído totalmente em P.

- Refinamento por falhas/divergências: O modelo de falhas não permite a fácil detecção de uma importante classe de estados: aquela em que o processo se encontra em *livelock* (por exemplo, uma sequência infinita de escolhas internas, assim o processo nunca executa um evento visível).

  Logo, um modelo mais minucioso que o de falhas é necessário. O modelo por falhas/divergências atende este requisito adicionando o conceito de divergências. As divergências de um processo são o conjunto de traces que ocorrem após o acontecimento de um *livelock.*

  As divergências trazem melhorias: tem-se a capacidade de analisar sistemas que não executam um evento visível, podendo afirmar que esses eventos não ocorrem em situações convenientes.

2.3.1. Objetos de FDR

Após compilado o arquivo CSP_M (versão legível por máquinas do CSP), o FDR é capaz de prover 4 funcionalidades principais [9]:

- *Session Management:* Responsável por carregar arquivos CSP e permitir a compilação do mesmo como um LTS [9];

- *Interpreted State Machines* (ISM): Representa a máquina de estados compilada anteriormente. É o núcleo central do funcionamento do FDR*:* permite a verificação do refinamento de LTSs compilados a partir de uma especificação CSP. O objeto *ISM* implementa três funcionalidades sobre o LTS: descrição (nome, apelido), estrutura (transições, alfabeto, eventos) e análise (refinamentos, livre de livelock, livre de deadlock);

- *Hypothesis*: Responsável pela checagem do refinamento compilado via *ISM*, o objeto *Hypothesis* gera um relatório de sucesso ou um *debugging information* em caso de falha;

- *Debugging Information*: Uma descrição detalhada do causador da falha do refinamento, que descreve três aspectos: contexto da falha, árvore da falha e comportamento dos nós LTS e seus filhos.

A Figura 1 [7] apresenta os métodos contidos nos objetos explicitados anteriormente. O objeto *Session* tem como funcionalidade principal compilar um modelo CSP escolhido gerando um LTS, porém possui outros métodos como a listagem de *assertions* (afirmações a serem checadas pelo FDR).

O objeto *ISM* possui métodos que acessam a estrutura do LTS gerado, como transições, alfabeto, nó raiz, descrição, obtenção do nome de um evento através do seu número identificador, entre outros. Além da estrutura, métodos adicionais relativos ao comportamento, testes se o modelo LTS é livre de deadlocks, livelocks, se é determinístico, até o refinamento entre dois LTS, processo explicado em detalhes anteriormente.

O objeto *Hypothesis* possui métodos de verificação das afirmações feitas anteriormente no ISM (como, por exemplo, a checagem do determinismo de um LTS). Os métodos de verificação são acompanhados da

geração de um objeto *Debug*, que pode ser utilizado para obter mais informações sobre a possível falha encontrada. O objeto *Debug* possui métodos que descrevem com mais detalhes a falha na verificação feita; por exemplo, como os processos envolvidos e a geração de uma árvore de erro.

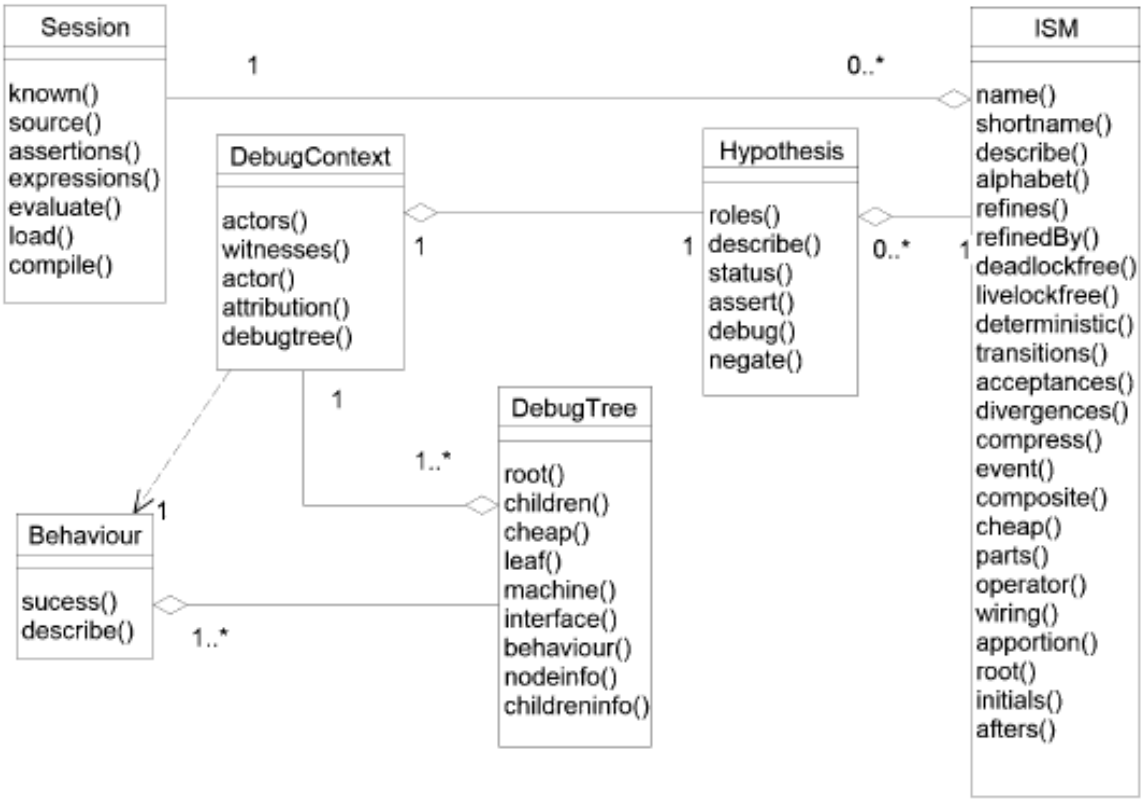


Figura 1 - Diagrama UML de FDR

3. NAT2TEST e Critérios de Teste

3.1. NAT2TEST

NAT2TEST (*Test Case Generation from Natural Language Requirements*) [1] é uma ferramenta desenvolvida no contexto de uma parceria entre a Embraer e o Centro de Informática da UFPE. Esta foi projetada com o intuito de automatizar a geração de casos de teste. Um dos aspectos fundamentais da NAT2TEST é o seu formato de entrada. Requisitos descritos em linguagem natural controlada são processados pela ferramenta para gerar casos de teste.

A facilidade de definir e editar os requisitos e, automaticamente, re-gerar os casos de testes é um dos pontos fortes da ferramenta. Além disso, a NAT2TEST dispõe de recursos que auxiliam o processo de criação dos casos de teste.

Uma especificidade da ferramenta é a capacidade de representação da passagem de tempo, o que permite descrever requisitos temporais. Esta característica é descrita com detalhes em [11].

O funcionamento da ferramenta é composto por cinco fases:

- Análise Sintática: Primeiramente, os requisitos são analisados sintaticamente de acordo com *SysReq-CNL* [1], uma Linguagem Natural Controlada [2] proposta para descrever DFRS [12] (*Data-Flow Reactive Systems*, uma caracterização formal para sistemas que possuem entrada e saída de dados, além da necessidade da representação de passagem de tempo);

- Análise Semântica: Na segunda fase, os requisitos são analisados semanticamente utilizando a teoria de Gramática de Casos. Nesse ponto

são gerados os *Case Frames,* uma estrutura que representa informalmente o significado semântico da sentença analisada [1].

- Geração do DFRS: Nesta fase, os *Case Frames* gerados anteriormente são analisados para a geração de um DFRS válido;

- Geração do CSP: O CSP que representa os requisitos iniciais é obtido a partir do DFRS gerado na etapa anterior [11]. Como já mencionado, outros modelos podem ser gerados, mas o foco deste trabalho é na cobertura da geração a partir de CSP

- Geração dos casos de teste: A geração de testes é feita com o auxílio das ferramentas FDR [7], através da geração de contraexemplos, e Z3 [10]. Esta última, uma ferramenta para resolver problemas SMT, cuja função na NAT2TEST é tratar numericamente as variáveis de tempo existentes no CSP (representadas apenas simbolicamente), analisando matematicamente a satisfatibilidade das equações geradas.

No contexto da geração automática de testes, a definição de critérios de cobertura é de suma importância para que os casos gerados possuam relevância em relação ao comportamento do sistema, ou seja, os testes sejam capazes de encontrar a grande maioria dos erros do sistema, e não sejam produzidos a livre demanda.

Para a ferramenta NAT2TEST, foram definidos e implementados neste trabalho cinco critérios (dois destes critérios possuem duas versões diferentes).

3.2. Geração Automática de Testes em NAT2TEST

A geração automática de testes na NAT2TEST é feita através das ferramentas FDR e Z3, como explicado no capítulo anterior. Porém, o escopo deste trabalho limita-se apenas a fase da geração de contraexemplos utilizando FDR.

Os casos de testes são gerados através contraexemplos resultantes dos refinamentos por *traces* entre dois processos CSP. Neste trabalho, um contraexemplo é considerado um caso de teste simbólico. Enquanto o primeiro processo representa o comportamento original do sistema, o segundo representa o processo original modificado com uma "marca".

Esta "marca" é utilizada para diferenciar parte do código CSP a fim de direcionar o refinamento FDR, forçando os contraexemplos gerados a passarem por certa parte do processo, pois os dois modelos CSP se diferenciam apenas neste aspecto.

Para continuar a identificação de contraexemplos, é criado um novo processo que é composto pela escolha externa entre o processo original e o contraexemplo gerado. Em seguida, verifica-se se este novo processo é refinado por *traces* com o processo marcado, surgindo um novo contraexemplo que também é utilizado para a geração dos próximos contraexemplos.

Todo este procedimento pode ser encontrado na Figura 2, que possui um exemplo básico da geração de contraexemplos em TCL, e a Figura 3, que mostra a implementação das funções "exChoice1" e "exChoice2" em CSP, utilizadas no código TCL.

```
1  set s [load "exemplo.csp"]
2  set S [$s compile S -t]
3  set Z [$s compile S_marcado -t]
4
5  set exChoice "exChoice1(S,exChoice2({"
6  set exChoiceClose "}))"
7
8  set hypothesis [$Z refines $S]
9  set assert [$hypothesis assert]
10 set traces ""
11
12 while {($assert == "false" || $assert == "xfalse")} {
13
14     set debug [$hypothesis debug]
15     set beh [$debug attribution 0 1]
16     set description [$beh describe]
17
18     set result [string map {"\f" " " "Performs\t" "" "\v" ""} $description]
19     set result [string map {"_tau " "" " " ","} $result]
20
21     append traces $result
22     set resp $exChoice$traces$exChoiceClose
23
24     append traces ","
25
26     set T [$s compile $resp -t]
27     set hypothesis [$Z refines $T]
28     set assert [$hypothesis assert]
29
30 }
```

Figura 2 - Código base da Geração de Contraexemplos

Primeiramente (linha 1), é carregado o modelo CSP que possui os processos a serem utilizados, o que representa o comportamento original do sistema e o modificado com a “marca”.

Após (linhas 2 e 3), os processos S (comportamento original) e S_marcado (processo com a “marca”) são transformados em objetos ISM de FDR. Nas linhas 5 e 6 são inicializadas as variáveis que carregam a chamada dos métodos CSP “exChoice1” e “exChoice2”, responsáveis pela escolha externa entre o processo original e os contraexemplos gerados, afim de continuar a geração de contraexemplos.

Na Linha 7, um objeto *Hypothesis* de FDR é criado, com o refinamento entre o processo S e S_marcado, posteriormente, é feito o teste se o refinamento ocorreu com sucesso.

O loop da geração de contraexemplos é iniciado, este irá continuar até que o FDR retorne um refinamento que seja realizado com sucesso (sem um trace de contraexemplo). Nas linhas 14, 15 e 16, um objeto *Debug* de FDR é criado e o contraexemplo do refinamento é obtido.

As linhas 17 e 18 são responsáveis por formatar a string do contraexemplo retornado para um formato conveniente a ser utilizado pelas funções “exChoice1” e “exChoice2”. O contraexemplo formatado é concatenado com as variáveis exChoice e exChoiceClose (linha 22), formando uma string no formato “exChoice1(S,exChoice2({contraexemplo_formatado}))”.

Por fim, um objeto ISM da escolha externa é criado, e o refinamento entre este processo e o processo marcado é feito, continuando a geração de contraexemplos.

```
1  exChoice1(PROC1, PROC2) = ( PROC1 [] PROC2 )
2  exChoice2(X) = [] p : X @ Proc(p)
3  Proc(<>) = STOP
4  Proc(s) = head(s) -> Proc(tail(s))
```

Figura 3 - Funções CSP exChoice1 e exChoice2

O processo de escolha externa utilizado na geração dos contraexemplos é feito utilizando os métodos CSP da Figura 3. A função “exChoice1” faz uma escolha externa simples, entre o processo original (que representa o comportamento do sistema, sem nenhuma modificação) e o contraexemplo gerado.

Porém, o contraexemplo necessita de um tratamento de formato para ser utilizado, feito na linha 2, através da função “exChoice2”, que por sua vez utiliza a função “Proc”, responsável por transformar os contraexemplos em um processo CSP.

3.3. Critérios de Cobertura

Foram implementados cinco critérios de cobertura, que, quando alcançados, determinam a parada da geração de contraexemplos:

- Quantidade de contraexemplos: Gera uma quantidade de contraexemplos (testes) definida pelo usuário;
- Transições | Caminhos: Gera contraexemplos até que todas as transições/caminhos sejam percorridas. Implementado de duas maneiras;
    a. Faz a contagem das transições percorridas levando em consideração os caminhos gerados pelos contraexemplos;
    b. Faz a contagem das transições percorridas levando em consideração apenas os eventos CSP contidos nos contraexemplos;
- Nós | Estados: Gera contraexemplos até que todos os nós/estados sejam percorridos. Implementado de duas maneiras:
    a. Faz a contagem dos nós percorridos levando em consideração os caminhos gerados pelos contraexemplos;
    b. Faz a contagem dos nós percorridos levando em consideração apenas os eventos CSP contidos nos contraexemplos;
- Ciclos Temporais: Gera X contraexemplos onde cada um possui Y ciclos temporais (evento CSP inerente à estratégia NAT2TEST que representa a passagem de um ciclo de tempo), onde X e Y são números definidos pelo usuário;
- Requisitos: Gera X contraexemplos para cada requisito originalmente escrito em linguagem natural, onde X é um número definido pelo usuário.

3.4. Implementação

3.4.1. Quantidade de Contraexemplos

Este é o critério mais simples. Ele gera uma quantidade de contraexemplos definida pelo usuário, sem nenhum tipo de direcionamento, apenas seguindo as regras internas de geração de contraexemplos de FDR. Na Figura 4 é possível ver o código em uma linguagem genérica para melhor entendimento Todos os objetos FDR utilizados foram anteriormente explicados.

```
int qtdContraExemplos = UserInput();

csp_original = Session.load("original.csp");
csp_com_marca = Session.load("marcado.csp");

ISM processo_original = Session.compile(csp_original, 'traces');
ISM processo_marcado = Session.compile(csp_com_marca, 'traces');

Hypothesis refinamento_por_traces = processo_marcado.refines(processo_original);
Boolean resultado_refinamento = refinamento_por_traces.getResult();

while(qtdContraExemplos > 0 && resultado_refinamento == false){

    Debug refinamento_debug = refinamento_por_traces.getDebugObject();
    String[] contra_exemplo = refinamento_debug.getCounterExemple();

    print(contra_exemplo);
    qtdContraExemplos = qtdContraExemplos - 1;

    ISM processo_loop = Session.compile(csp_original + contra_exemplo, 'traces');
    refinamento_por_traces = processo_modificado.refines(processo_loop);
    resultado_refinamento = refinamento_por_traces.getResult();

}
```

Figura 4 - Código do critério Quantidade de Contraexemplos

Primeiramente, é esperada uma entrada de usuário que indicará a quantidade de contraexemplos a ser gerada. Logo após, dois arquivos CSP são carregados, um que representa o sistema com seu comportamento original e outro que representa o sistema com uma “marca” adicional, ideia já mencionada anteriormente.

Nas linhas 6 e 7, dois objetos FDR ISM foram criados a partir dos CSP carregados, com a opção de refinamento por traces. Posteriormente um objeto *Hypothesis* é criado, resultante do refinamento entre os dois ISM.

Inicia-se o loop que gerará os contraexemplos. Este rodará enquanto a variável "qtdContraExemplos" for maior que zero, sendo decrementada a cada execução do while, ou se algum refinamento interior ao loop retornar verdadeiro. Caso que acontece quando FDR já enumerou todos os contraexemplos existentes.

Para obter o contraexemplo do refinamento feito, utiliza-se um objeto *Debug* obtido a partir do objeto *Hypothesis*. Como visto anteriormente, o objeto Debug possui informações acerca do refinamento que não deu certo, entre estas, um contraexemplo.

Com o contraexemplo em mãos, inicia-se o processo de geração do próximo teste, que será feito utilizando um novo processo, representado pela soma entre o CSP original e o contraexemplo gerado (na prática, esta soma é feita utilizando uma escolha externa de FDR).

3.4.2. Transições | Caminhos

Este critério é um pouco mais complexo que o anterior, e foi implementado de duas maneiras diferentes. As transições são retornadas pelo FDR no seguinte formato: {0 1 1}, onde o primeiro e o último número representam os nós inicial e final da transição, e o número central representa o evento que liga os dois nós.

No primeiro modo em que o critério foi implementado, a contagem das transições percorridas é feita através dos caminhos gerados pelos contraexemplos, iniciando do nó raiz do LTS.

O fluxo básico se assemelha ao critério “quantidade de contraexemplos”, com a adição de um loop que percorre o LTS utilizando os eventos contidos no contraexemplo.

Variáveis que representam as transições e as transições percorridas foram criadas. O caminho inicia na linha 23 (Figura 5), onde o valor do nó raiz do LTS é atribuído à variável “firstNode”. Assim, é procurado na lista de transições se existe alguma que possua como nó inicial a raiz do LTS e, como evento, o primeiro evento contido no contraexemplo gerado.

```
1  csp_original = Session.load("original.csp");
2  csp_com_marca = Session.load("marcado.csp");
3
4  //função recebe o nome de um evento e retorna seu ID
5  function int getEvent(String eventName){};
6
7  ISM processo_original = Session.compile(csp_load, 'traces');
8  ISM processo_marcado = Session.compile(csp_com_marca, 'traces');
9
10 List transicoesPercorridas = {};
11 List transicoes = processo_original.getTransitions();
12 int qtdTransicoes = transicoes.lenght();
13
14 Hypothesis refinamento_por_traces = processo_modificado.refines(processo_original);
15 Boolean resultado_refinamento = refinamento_por_traces.getResult();
16
17 while(qtdTransicoes > 0 && resultado_refinamento == false){
18
19     Debug refinamento_debug = refinamento_por_traces.getDebugObject();
20     String[] contra_exemplo = refinamento_debug.getCounterExemple();
21     print(contra_exemplo);
22
23     int firstNode = processo_original.root();
24     for (i = 0; i < contra_exemplo.lenght(); i++) {
25
26         var transicao;
27         int evento = getEvent(contra_exemplo[i]);
28
29         if(transicao = transicoes.find(transicoesElement => transicoesElement.firstNode == firstNode
30         && transicoesElement.event == evento)){
31             firstNode = transicao.secondNome;
32         } else {
33             break;
34         }
35
36         if(!transicoesPercorridas.exists(percorrido => percorrido.firstNode == transicao.firstNode
37         && percorrido.event == evento
38         && percorrido.secondNode == transicao.secondNode)){
39             transicoesPercorridas.add(transicao);
40             qtdTransicoes = qtdTransicoes - 1;
41         }
42     }
43
44     ISM processo_loop = Session.compile(csp_original + contra_exemplo, 'traces');
45     refinamento_por_traces = processo_modificado.refines(processo_loop);
46     resultado_refinamento = refinamento_por_traces.getResult();
47
48 }
```

Figura 5 – Código do critério Transições modo 1

Se existir alguma transição com estas características, a variável “fistNode” recebe o valor do segundo nó desta transição, fazendo com que a próxima transição percorrida possua como primeiro nó o segundo nó da transição atual.

Após estes passos, a transição percorrida é inserida na linha de transições percorridas e laço prossegue.

No segundo modo de implementação deste critério de cobertura, encontrado na Figura 6, a contagem é feita levando em consideração o evento de cada transição, portanto todas as transições que possuírem determinado evento contido na lista de eventos do contraexemplo gerado serão percorridas de uma vez, não importando os nós de origem ou destino.

```
csp_original = Session.load("original.csp");
csp_com_marca = Session.load("marcado.csp");

//função recebe o nome de um evento e retorna seu ID
function int getEvent(String eventName){};

ISM processo_original = Session.compile(csp_original, 'traces');
ISM processo_marcado = Session.compile(csp_com_marca, 'traces');

List eventosPercorridos = {};
List transicoes = processo_original.getTransitions();
int qtdTransicoes = transicoes.lenght();

Hypothesis refinamento_por_traces = processo_marcado.refines(processo_original);
Boolean resultado_refinamento = refinamento_por_traces.getResult();

while(qtdTransicoes > 0 && resultado_refinamento == false){

    Debug refinamento_debug = refinamento_por_traces.getDebugObject();
    String[] contra_exemplo = refinamento_debug.getCounterExemple();
    print(contra_exemplo);

    for (i = 0; i < contra_exemplo.lenght(); i++) {

        int evento = getEvent(contra_exemplo[i]);

        if(!eventosPercorridos.exists(evento)){
            List transicoesEvento = transicoes.get(transicaoElement => transicaoElement.event == evento);
            qtdTransicoes = qtdTransicoes - transicoesEvento.lenght();
            eventosPercorridos.add(transicoesEvento.firstElement().event);
        }
    }

    ISM processo_loop = Session.compile(csp_original + contra_exemplo, 'traces');
    refinamento_por_traces = processo_modificado.refines(processo_loop);
    resultado_refinamento = refinamento_por_traces.getResult();

}
```

Figura 6 - Código critério Transições modo 2

Portanto, a principal diferença entre os modos 1 e 2 é o jeito como as transições são percorridas. No modo 2, todas as transições que possuem determinado evento são eliminadas de uma vez, logo este modo gera menos testes que o anterior.

3.4.3. Nós | Caminhos

Este critério em muito se parece com o explicitado anteriormente. Ocorre a geração de contraexemplos até que todos os nós do LTS sejam percorridos. Passar por todos os nós não necessariamente significa passar por todas as transições.

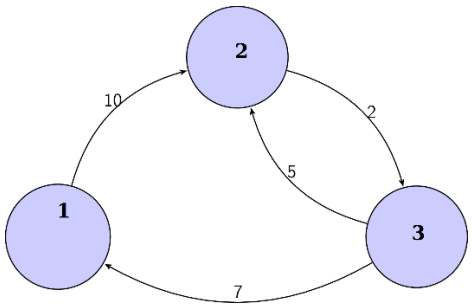


Figura 7 - Exemplo de LTS

Considerando o exemplo de LTS apresentado na Figura 7, podemos fazer o caminho {1 10 2}, {2 2 3} (seguindo a notação de transições de FDR), que percorre todos os nós, porém não faz todas as transições possíveis.

Este critério também foi criado de duas formas, seguindo os moldes do critério “Transições | Caminhos”, onde a primeira implementação leva em consideração o caminho gerado pelo contraexemplo, e a segunda não considera este fato.

O primeiro modo, mostrado na Figura 8, possui uma lista chamada “nosPercorridos”. Toda vez que um novo nó for encontrado, este é

adicionado na lista. O modo como o LTS é percorrido segue a mesma ideia do critério “Transições | Caminhos”.

```
csp_original = Session.load("original.csp");
csp_com_marca = Session.load("marcado.csp");

//função recebe o nome de um evento e retorna seu ID
function int getEvent(String eventName){};
//funcao recebe uma lista de transicoes e retorna a quantidade total de nós
function int getQtdNos(List transicoes){};

ISM processo_original = Session.compile(csp_original, 'traces');
ISM processo_marcado = Session.compile(csp_com_marca, 'traces');

List nosPercorridos = {};
List transicoes = processo_original.getTransitions();
int qtdNos = getQtdNos(transicoes);

Hypothesis refinamento_por_traces = processo_marcado.refines(processo_original);
Boolean resultado_refinamento = refinamento_por_traces.getResult();

while(qtdNos > 0 && resultado_refinamento == false){
    Debug refinamento_debug = refinamento_por_traces.getDebugObject();
    String[] contra_exemplo = refinamento_debug.getCounterExemple();
    print(contra_exemplo);

    int firstNode = processo_original.root();
    for (i = 0; i < contra_exemplo.lenght(); i++) {
        var transicao;
        int evento = getEvent(contra_exemplo[i]);
        if(!nosPercorridos.exist(firstNode)){
            nosPercorridos.add(firstNode);
            qtdNos = qtdNos - 1;
        }
        if(transicao = transicoes.find(transicoesElement => transicoesElement.firstNode == firstNode
        && transicoesElement.event == evento)){
            firstNode = transicao.secondNode;
        } else {
            break;
        }
    }
    ISM processo_loop = Session.compile(csp_original + contra_exemplo, 'traces');
    refinamento_por_traces = processo_modificado.refines(processo_loop);
    resultado_refinamento = refinamento_por_traces.getResult();
}
```

Figura 8 - Código critério Nós | Caminhos modo 1

O segundo modo, exibido na Figura 9, segue a mesma linha do modo dois do critério “Transições | Caminhos”, com a diferença da checagem dos nós percorridos ao invés das transições.

```
csp_original = Session.load("original.csp");
csp_com_marca = Session.load("marcado.csp");

//função recebe o nome de um evento e retorna seu ID
function int getEvent(String eventName){};
//funcao recebe uma lista de transicoes e retorna a quantidade total de nós
function int getQtdNos(List transicoes){};

ISM processo_original = Session.compile(csp_original, 'traces');
ISM processo_marcado = Session.compile(csp_com_marca, 'traces');

List nosPercorridos = {};
List eventosPercorridos = {};
List transicoes = processo_original.getTransitions();
int qtdNos = getQtdNos(transicoes);

Hypothesis refinamento_por_traces = processo_marcado.refines(processo_original);
Boolean resultado_refinamento = refinamento_por_traces.getResult();

while(qtdNos > 0 && resultado_refinamento == false){
    Debug refinamento_debug = refinamento_por_traces.getDebugObject();
    String[] contra_exemplo = refinamento_debug.getCounterExemple();
    print(contra_exemplo);

    for (i = 0; i < contra_exemplo.lenght(); i++) {
        int evento = getEvent(contra_exemplo[i]);
        if(!eventosPercorridos.exists(evento)){
            List transicoesEvento = transicoes.get(transicaoElement => transicaoElement.event == evento);
            for (j = 0; i < transicoesEvento.lenght(); j++) {
                if(!nosPercorridos.exist(transicoesEvento[j].firstNode){
                    nosPercorridos.add(transicoesEvento[j].firstNode);
                    qtdNos = qtdNos - 1;
                }
                if(!nosPercorridos.exist(transicoesEvento[j].secondNode){
                    nosPercorridos.add(transicoesEvento[j].secondNode);
                    qtdNos = qtdNos - 1;
                }
            }
            eventosPercorridos.add(transicoesEvento.firstElement().event);
        }
    }
    ISM processo_loop = Session.compile(csp_original + contra_exemplo, 'traces');
    refinamento_por_traces = processo_modificado.refines(processo_loop);
    resultado_refinamento = refinamento_por_traces.getResult();
}
```

Figura 9 – Código critério Nós | Caminhos modo 2

3.4.4. Ciclos Temporais

O critério de ciclos temporais possui uma implementação simples, porém sua marcação é um pouco complexa. O objetivo é gerar uma quantidade X de contraexemplos na qual cada contraexemplo possua Y ciclos temporais, onde X e Y são números definidos pelo usuário. O código de como a marcação é feita pode ser encontrado na Figura 10:

```
1  function ciclosTemporaisMarcacao(i, y){
2      esperarEventoPassagemTempo();
3      if(i == y) {
4          inserirMarcacao();
5      } else {
6          ciclosTemporaisMarcacao(i+i, y);
7      }
8  }
```

Figura 10 - Código da função de marcação do critério Ciclos Temporais

A ideia central da função “ciclosTemporaisMarcacao” é forçar o CSP a esperar uma quantidade Y de ciclos temporais antes de inserir a marcação; ou seja, se a quantidade de eventos de passagem de tempo for igual a Y, a marcação é inserida, se não, a função é chamada recursivamente com um incremento no contador.

O CSP modificado é inicializado através desta função (linha 5 da Figura 11), que irá alterar o comportamento descrito, o evento de marcação será lançado após uma quantidade de ciclos temporais determinada pelo usuário. Esta marcação é originalmente feita dentro do próprio CSP, mostrado no Apêndice A.

Após a marcação ter sido inserida, o código de geração dos testes fica muito parecido com o critério “quantidade de contraexemplos”. Na Figura 11 é possível ver o código.

```
int qtdContraExemplos = UserInput();
int qtdCiclosTemporais = UserInput();

csp_original = Session.load("original.csp");
csp_com_marca = Session.load("marcadoCiclosTemporais.csp", ciclosTemporaisMarcacao(0, qtdCiclosTemporais));

ISM processo_original = Session.compile(csp_original, 'traces');
ISM processo_marcado = Session.compile(csp_com_marca, 'traces');

Hypothesis refinamento_por_traces = processo_marcado.refines(processo_original);
Boolean resultado_refinamento = refinamento_por_traces.getResult();

while(qtdContraExemplos > 0 && resultado_refinamento == false){

    Debug refinamento_debug = refinamento_por_traces.getDebugObject();
    String[] contra_exemplo = refinamento_debug.getCounterExemple();

    print(contra_exemplo);
    qtdContraExemplos = qtdContraExemplos - 1;

    ISM processo_loop = Session.compile(csp_original + contra_exemplo, 'traces');
    refinamento_por_traces = processo_modificado.refines(processo_loop);
    resultado_refinamento = refinamento_por_traces.getResult();

}
```

Figura 11 - Código critério Ciclos Temporais

Os códigos CSP e TCL deste critério podem ser encontrados na sessão de Apêndices deste trabalho (apêndices A).

3.4.5. Requisitos

Este critério tem como objetivo gerar uma quantidade X de testes para cada requisito descrito em linguagem natural, onde X é um número escolhido pelo usuário. Ele possui uma implementação relativamente simples, porém as marcações são relativamente complicadas.

Devido ao nível de complexidade, as marcações foram feitas utilizando a linguagem Java, através de uma análise sintática dos processos contidos no CSP. O processo que representa os requisitos do sistema além de todos os que se comunicam com ele são modificados.

Cada requisito necessita de uma marcação diferente, logo, é gerado um novo processo para cada requisito descrito (além dos que se comunicam com ele, que também serão replicados).

O código de como é feita a marcação pode ser encontrada na Figura 12:

```
1  function requisitosMarcacao(String csp){
2
3      String processoRequisitos = getProcessoRequisitos(csp);
4      String nomeProcessoRequisitos = getNomeProcesso(processoRequisitos);
5      int qtdRequisitos = getQuantidadeRequisitos(processoRequisitos);
6
7      while(qtdRequisitos > 0){
8
9          List conteudoNovoProcessoRequisitos = {};
10         conteudoNovoProcessoRequisitos.add(csp);
11
12         String novoNomeProcessoRequisitos = nomeProcessoRequisitos + qtdRequisitos;
13         //marcarProcessoRequisito(String requisitos, int n_requisito)
14         //marca o requisito indicado no número e o retorna
15         String novoProcessoRequisitos = marcarProcessoRequisito(processoRequisitos, qtdRequisitos);
16         conteudoNovoProcessoRequisitos.add(novoProcessoRequisitos);
17
18         //getProcessosInfluenciados(String processo) retorna uma lista com todos os processos
19         //que se comunicam com o processo dado como entrada
20         String[] processosInfluenciados = getProcessosInfluenciados(nomeProcessoRequisitos);
21         String[] processosInfluenciadosNomes = getNomesProcessosInfluenciodos(processosInfluenciados);
22
23         for (int i = 0; i < processosInfluenciados.lenght(); i++) {
24             String nomeAntigo = processosInfluenciadosNomes[i];
25             String nomeNovo = nomeAntigo + qtdRequisitos;
26             processosInfluenciados[i].replaceAll(nomeAntigo, nomeNovo);
27             processosInfluenciados[i].replaceAll(nomeProcessoRequisitos, novoNomeProcessoRequisitos);
28             conteudoNovoProcessoRequisitos.add(processosInfluenciados[i]);
29         }
30
31         createFile("requisito" + qtdRequisitos + ".csp", conteudoNovoProcessoRequisitos);
32     }
33 }
```

Figura 12 - Código da função de marcação do Critério Requisitos

Primeiramente são obtidos do CSP, nas linhas 3 e 4, o conteúdo do processo que representa o comportamento dos requisitos e o nome deste. Logo após, a quantidade de requisitos é obtida analisando o conteúdo do processo armazenado anteriormente.

Inicia-se um laço, que irá gerar para cada requisito um novo arquivo (linha 31) que representará o novo processo marcado, os processos que se comunicam com este também modificados, mais o antigo conteúdo já contido no CSP original.

Por exemplo, seja o processo `P` aquele que possui a descrição dos requisitos do sistema, e os processos `Q` e `R` os que se comunicam direta ou indiretamente com P:

```
P = eventos_internos -> (
(requisito_1 -> SKIP)[]
(requisito_2 -> SKIP)[]
)

Q = P -> evento_generico -> SKIP

R = Q -> evento_generico_2 -> SKIP
```

Após o processamento em Java, descrito na Figura 12, surgem os seguintes novos processos, sendo os primeiros para o requisito 2, e os posteriores para o requisito 1:

- Requisito 2

```
P_2 = eventos_internos -> (
(requisito_1 -> SKIP)[]
(requisito_2 -> marcação -> SKIP)[]
)

Q_2 = P_2 -> evento_generico -> SKIP

R_2 = Q_2 -> evento_generico_2 -> SKIP
```

- Requisito 1

```
P_1 = eventos_internos -> (
(requisito_1 -> marcação -> SKIP)[]
```

```
(requisito_2 -> SKIP)[]
)

Q_1 = P_1 -> evento_generico -> SKIP

R_1 = Q_1 -> evento_generico_2 -> SKIP
```

Com a marcação feita, é possível utilizar o FDR para gerar contraexemplos utilizando o CSP original e todos os que foram criados via Java (através do método da Figura 12).

A implementação da geração de testes é simples. São gerados X contraexemplos (X é um número definido pelo usuário) para cada CSP marcado, gerado anteriormente. O código pode ser acompanhado abaixo:

```
1  int qtdContraExemplos = UserInput();
2  int qtdRequisitos = JavaInput();
3
4  csp_original = Session.load("original.csp");
5  ISM processo_original = Session.compile(csp_original, 'traces');
6
7  while(qtdRequisitos > 0){
8
9      String nome_csp_marcado = "requisito" + qtdRequisitos + ".csp";
10     csp_com_marca = Session.load(nome_csp_marcado);
11
12     ISM processo_marcado = Session.compile(csp_com_marca, 'traces');
13
14     Hypothesis refinamento_por_traces = processo_marcado.refines(processo_original);
15     Boolean resultado_refinamento = refinamento_por_traces.getResult();
16
17     while(qtdContraExemplos > 0 && resultado_refinamento == false){
18
19         Debug refinamento_debug = refinamento_por_traces.getDebugObject();
20         String[] contra_exemplo = refinamento_debug.getCounterExemple();
21
22         print(contra_exemplo);
23         qtdContraExemplos = qtdContraExemplos - 1;
24
25         ISM processo_loop = Session.compile(csp_original + contra_exemplo, 'traces');
26         refinamento_por_traces = processo_modificado.refines(processo_loop);
27         resultado_refinamento = refinamento_por_traces.getResult();
28
29     }
30 }
```

Figura 13 - Código critério Requisitos

A quantidade de requisitos contidos no CSP é recebida via Java, no momento da inicialização do script de geração de testes. A cada loop é feito o refinamento entre o CSP original e o CSP modificado referente ao requisito numerado pela variável “qtdRequisitos”.

Ao término da geração, inicia-se um novo loop, carregando um novo CSP modificado referente ao próximo requisito.

3.5. Análise Empírica

Os processos CSP analisados por FDR por vezes geram modelos LTS complexos, resultando na demora para executar os critérios implementados. Nesta sessão encontram-se análises que podem auxiliar algum futuro estudo relativo a melhora de performance e quantidade dos contraexemplos gerados.

3.5.1. Complexidade do LTS

Um modo de medir a complexidade de um modelo LTS pode ser a partir da quantidade de nós e transições que possui. Alguns critérios deste trabalho são diretamente influenciados por estes fatores e outros indiretamente.

Os critérios “Nós | Caminhos” e “Transições | Caminhos” são influenciados diretamente, os outros são indiretamente afetados, por conta de processamentos feitos por FDR para fazer os refinamentos que geram os contraexemplos. Todas as medições foram feitas utilizando o mesmo processo CSP base.

Abaixo, a Tabela 1 apresenta um resumo dos critérios versus quantidade de transições e nós, utilizando um processo CSP gerado por NAT2TEST.

Critério	Quantidade Transições	Quantidade Nós
Nós, Transições e Quantidade de Contraexemplos	1797	1710
Ciclos Temporais (um ciclo temporal por contraexemplo)	7756	7193
Requisitos (um contraexemplo por requisito)	?	?

Tabela 1 - Quantidade de transições e nós

Os critérios Nós, Transições e Quantidade de Contraexemplos são baseados no mesmo processo CSP (as marcas são colocadas no mesmo local), portanto possuem a mesma quantidade de transições e nós.

A marca utilizada no critério “ciclos temporais” é mais complexa, acarretando em um LTS com quatro vezes mais nós e transições. O critério de requisitos se mostrou tão complexo que não foi possível calcular a quantidade de nós e transições em tempo hábil (escala de horas).

3.5.2. Tempo versus Contraexemplos

Outra maneira de analisar a performance dos critérios implementados, é através comparação entre tempo decorrido e a quantidade de contraexemplos gerados até então. Um gráfico foi gerado para cada critério implementado:

- Critério Quantidade de contraexemplos

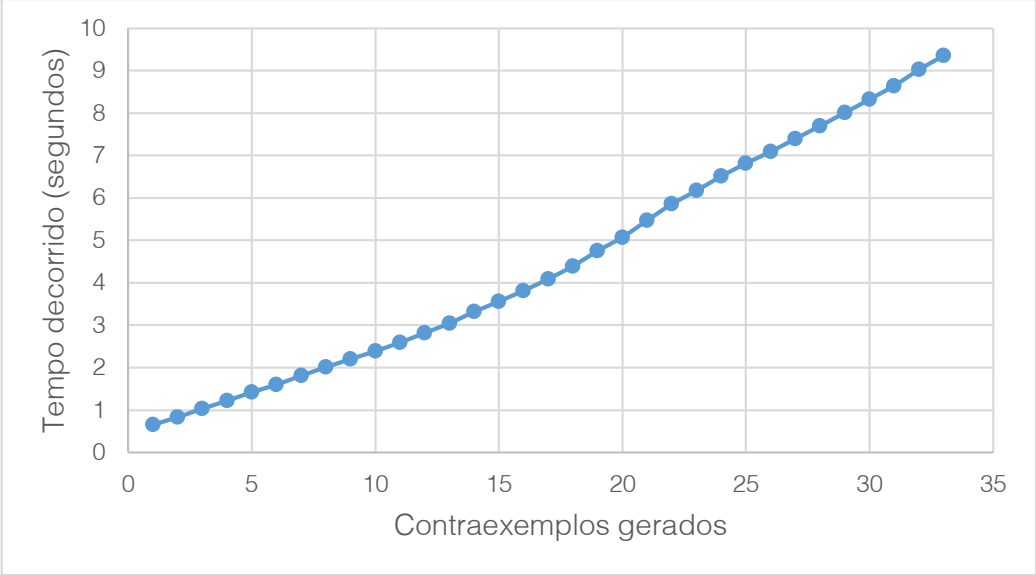


Gráfico 1 - Tempo x Contraexemplos, critério quantidade de contraexemplos

No Gráfico 1, a partir do trigésimo quarto contraexemplo, a escala de tempo para obter um novo caso de teste muda para horas (tempo exato não conhecido com precisão).

- Critério Nós | Caminhos modo 1

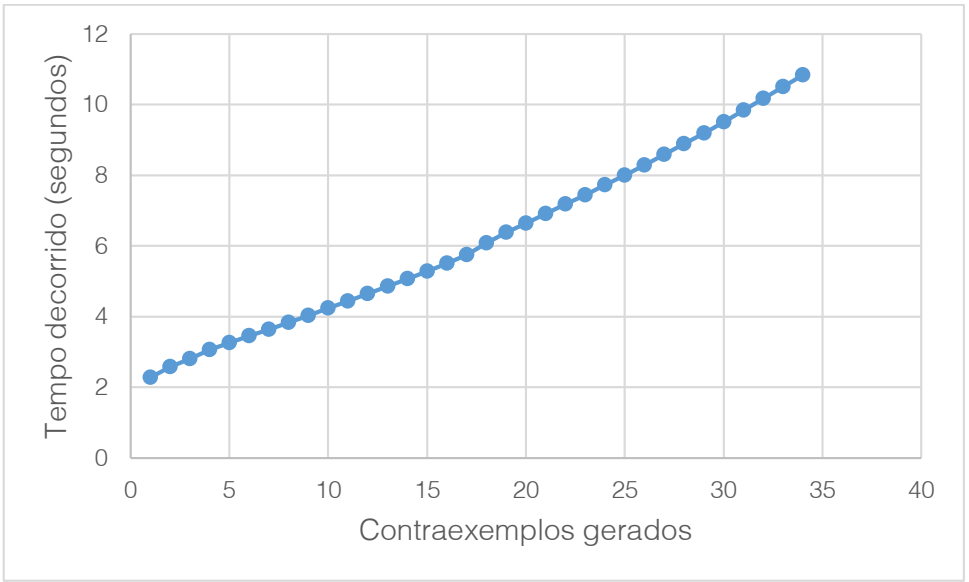


Gráfico 2 - Tempo x Contraexemplos, critério nós | caminhos modo 2

No Gráfico 2, a partir do trigésimo quinto contraexemplo, a escala de tempo para obter um novo caso de teste muda para horas (tempo exato não conhecido com precisão).

- Critério Nós | Caminhos modo 2

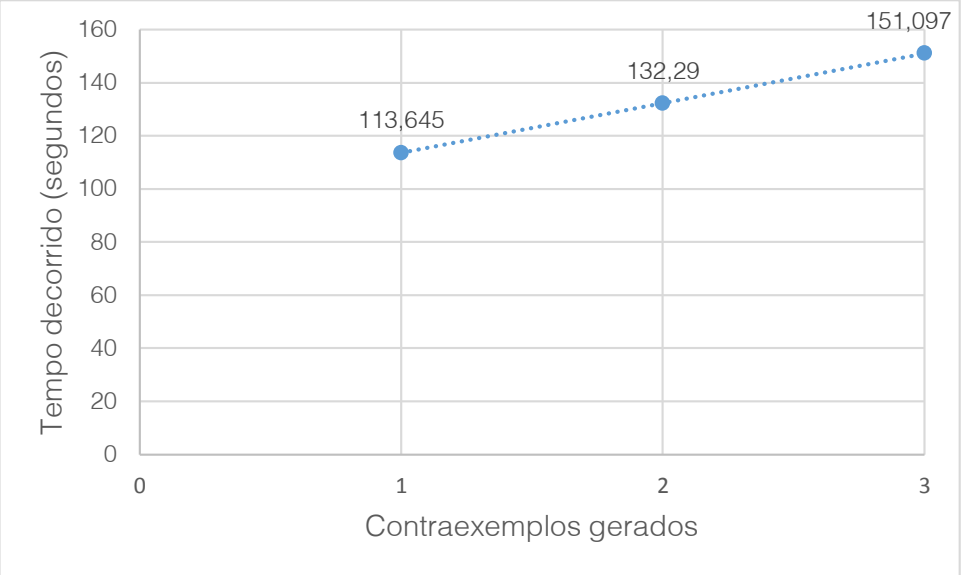


Gráfico 3 - Tempo x Contraexemplos, critério nós | caminhos modo 2

O critério "Nós | Caminhos modo 2" gerou apenas três contraexemplos, e finalizou com sucesso. O maior tempo do primeiro contraexemplo deve-se aos processamentos feitos antes dos refinamentos propriamente ditos.

- Critério Transições | Caminhos modo 1

No Gráfico 2, a partir do trigésimo quinto contraexemplo, a escala de tempo para obter um novo caso de teste muda para horas (tempo exato não conhecido com precisão).

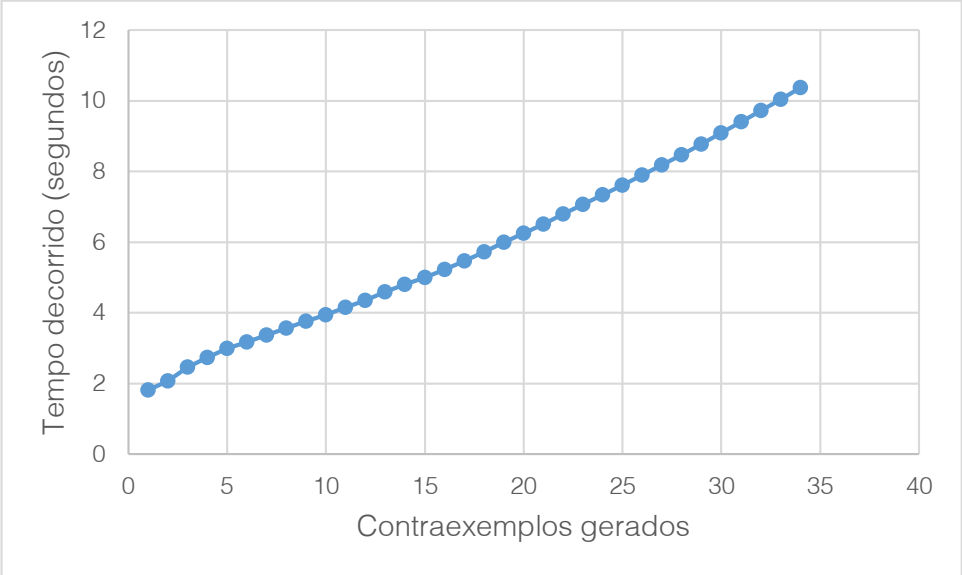


Gráfico 4- Tempo x Contraexemplos, critério transições | caminhos modo 1

- Critério Transições | Caminhos modo 2

O critério "Transições | Caminhos modo 2" gerou vinte e quatro contraexemplos e finalizou com sucesso, como visto no Gráfico 5. O intervalo entre os contraexemplos 5 e 17 são gerados mais rapidamente por possuírem muitas transições já percorridas.

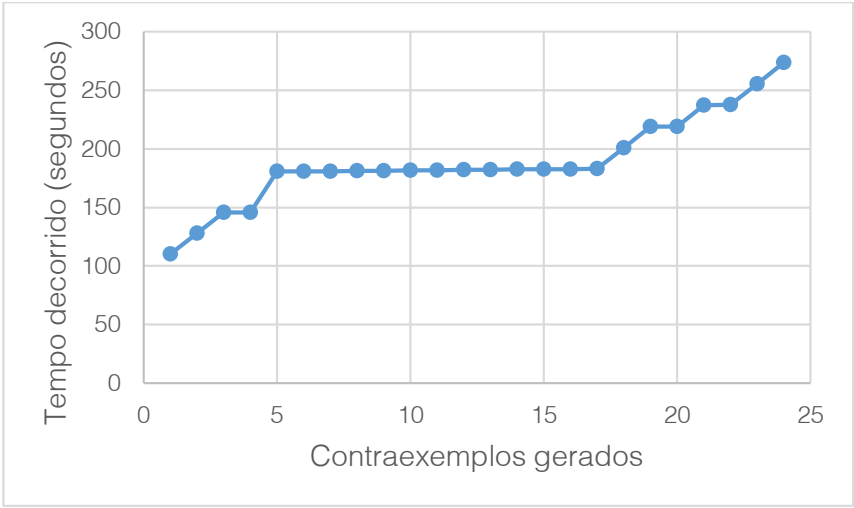
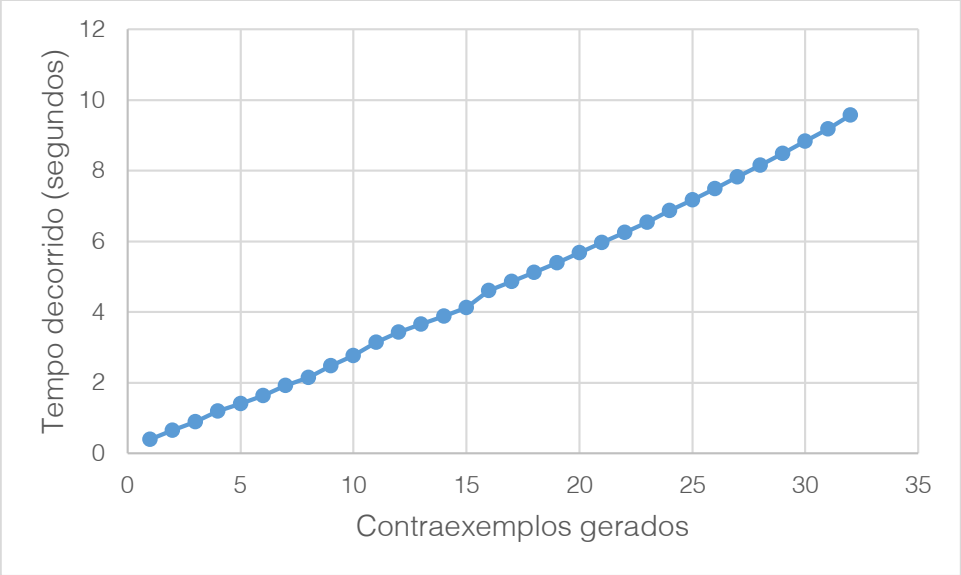


Gráfico 5 - Tempo x Contraexemplos, critério transições | caminhos modo 2

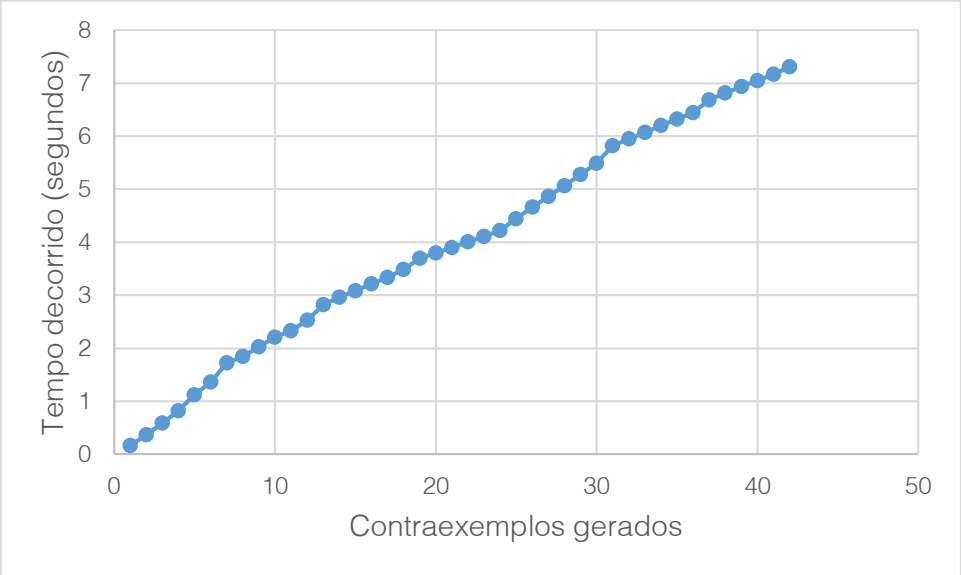
- Critério Ciclos Temporais:

No Gráfico 6, a partir do trigésimo terceiro contraexemplo, a escala de tempo para obter um novo caso de teste muda para horas (tempo exato não conhecido com precisão). Foram gerados contraexemplos com 1 ciclo temporal cada.



- Critério Requisitos

No Gráfico 7, a partir do quadragésimo segundo contraexemplo, a escala de tempo para obter um novo caso de teste muda para horas (tempo exato não conhecido com precisão). Foram gerados 6 contraexemplos para cada requisito descrito.


Tempo decorrido (segundos)
0
1
2
3
4
5
6
7
8
0
10
20
30
40
50
Contraexemplos gerados

4. Conclusão

O objetivo deste trabalho foi alcançado com sucesso, entretanto, alguns problemas surgiram durante a implementação dos critérios de cobertura de geração automática de testes.

Para exemplos de maior complexidade, o modelo CSP gerado pela ferramenta NAT2TEST é convertido por FDR para um LTS com muitos estados, o que faz com que alguns dos critérios implementados demorem a executar. Em particular, durante a geração de uma quantidade maior de contraexemplos para sistemas de maior complexidade, o tempo para compilar o código CSP superou a ordem de horas, o que limita o número de testes gerados pelos critérios implementados.

4.1. Trabalhos Relacionados

Estudos mostram que a etapa de testes de software, em muitos casos, corresponde a 50% do custo total de desenvolvimento [13]. Isto mostra a importância de métodos que melhorem e automatizem esta atividade, desde a concepção até sua execução.

Uma das abordagens existentes para a geração automática de testes é a MBT (*Model-Based Testing*), mencionada no Capítulo 2, da qual a NAT2TEST faz uso. Além da NAT2TEST, é possível encontrar outras ferramentas que utilizam o MTB:

- TaRGet: Ferramenta desenvolvida em uma parceria entre o Centro de Informática da UFPE e a Motorola, utiliza linguagem natural como entrada para a geração de testes.

  A TaRGet transforma os requisitos descritos em linguagem natural em casos de teste que possuem informações sobre a ação do

usuário, estado do sistema e a resposta para cada passo do caso de uso.

A geração de testes é feita de forma análoga a NAT2TEST, utilizando a linguagem CSP juntamente com FDR, porém o modelo utilizado por TaRGet para descrever o comportamento do sistema é diferente.

A TaRGet também possui critérios de geração de testes, trabalho feito por Leonardo em [14], que serviu como base para o desenvolvimento dos critérios no contexto da NAT2TEST. Foram implementados três critérios de cobertura: *MaxNumber,* gera testes até atingir o limite definido pelo usuário; *AllSteps*, gera testes até que cada passo de cada caso de uso seja coberto pelo menos uma vez e *AllTransitions,* gera testes até que cada transição do LTS seja percorrido pelo menos uma vez.

- Spec Explorer: Ferramenta desenvolvida pela Microsoft, sucessora da ferramenta AsmL Test Tool [16], utiliza a linguagem C# para descrever os modelos e a linguagem Cord na geração e propósitos de testes.

- Conformiq: Ferramenta utilizada para projetar e gerar casos de teste automaticamente. Utiliza como entrada modelos do sistema em Java e UML. Matematicamente calcula um conjunto de testes e os exporta em formatos como TCL, VB e Python.

As ferramentas NAT2TEST e TaRGet implementam coberturas de teste relacionadas a requisitos descritos em forma de modelos, existem outras que que executam os testes diretamente no código, estas implementam

cobertura de código. Na tabela 2, um comparativo entre funcionalidades existentes nas ferramentas mencionadas.

Nome	Organização	Entrada	Cobertura	CNL	Propósito de Teste
NAT2TEST	Centro de Informática – UFPE	Requisitos descritos em Linguagem Natural	Sim, opções variadas	Sim	Não
TaRGet	Centro de Informática – UFPE	Casos de uso descritos em Linguagem Natural	Sim, opções variadas	Sim	Sim
Spec Explorer	Microsoft	Modelos em C#	Sim	Não	Sim
Conformiq	Conformiq Software Limited	Java e Diagramas UML	Sim	Não	Não

Tabela 2 - Comparativo entre ferramentas de geração de testes

4.2. Trabalhos Futuros

- Integração com a ferramenta NAT2TEST: Os critérios foram implementados baseados em NAT2TEST, porém ainda não foram totalmente integrados a ferramenta.

- Quantidade de contraexemplos gerados: Estudar alternativas que permitam gerar uma maior quantidade de contraexemplos, quando o sistema considerado for de maior complexidade.

Referências

[1] CARVALHO, Gustavo H. P.; FALCÃO, Diogo; BARROS, Flavia; SAMPAIO, Augusto; MOTA, Alexandre; MOTTA, Leonardo; BLACKBURN, Mark. NAT2TEST_SCR: Test Case Generation from Natural Language Requirements based on SCR Specifications. Science of Computer Programming (Print), 2014.

[2] ALLEN, J. Natural Language Understanding, Benjamin/Cummings, 1995.

[3] SCHNEIDER, S. Concurrent and Real Time Systems: the CSP Approach, John Wiley, 1999.

[4] BASTOS, Anderson; RIOS, Emerson; CRISTALLI, Ricardo; MOREIRA, Trayahú; Base de Conhecimento em Teste de Software; São Paulo: Martins; 2007.

[5] TRETMANS, Jan; Model Based Testing with Labelled Transition Systems. Embedded Systems Institute, Eindhoven and Radboud University, Nijmegen, The Netherlands; 2008.

[6] ROSCOE, A. W. The Theory and Practice of Concurrency. Prentice Hall PTR, 1998.

[7] Formal Systems. Failures-Divergence Refinement - FDR2 User Manual. Formal Systems (Europe) Ltd, June 2005.

[8] JACKSON, Daniel; WING, Jeannette. "Lightweight Formal Methods", IEEE Computer, April 1996

[9] FREITAS, Leo; WOODCOCK, Jim. FDR Explorer. Department of Computer Science, University of York. YO10 5DD York, UK

[10] Z3 – Home – Codeplex. Disponível em: http://z3.codeplex.com/

[11] A.West, NASA Study on Flight Software Complexity, Tech. rep., NASA, 2009.

[12] CARVALHO, Gustavo H. P.; SAMPAIO, Augusto; MOTA, Alexandre. A CSP Timed Input-Output Relation and a Strategy for Mechanised Conformance Verification. In: International Conference on Formal Engineering Methods, 2013, Queenstown. Proceedings of the International Conference on Formal Engineering Methods, 2013.

[13] RAMLER, R. and WOLFMAIER, K. Economic perspectives in test automation – balancing automated and manual testing with opportunity cost. Workshop on Automation of Software Test ICSE, 2006.

[14] LEONARDO, Hugo. Extensão da ferramenta TaRGet para geração automatizada de casos de teste a partir do uso de variáveis. Trabalho de Conclusão de Curso – Centro de Informática UFPE, Recife, 2013.

[15] Spec Explorer (2010). Disponível em: http://research.microsoft.com/en-us/projects/specexplorer/

[16] BARNETT, M.; GRIESKAMP, W.; NACHMANSON, L.; SCHULTE, W.; TILLMANN, N. and VEANES, M. Towards a tool environment for model-based testing with asml. Microsoft Research, 2004.

[17] Conformiq (2010). Conformiq test generator. http://www.conformiq.com/products.php, 2014.

5. Apêndices

Apêndice A – Código CSP e TCL, método de marcação do critério Ciclos Temporais e critério Ciclos Temporais

```
1  Q(i,y) = time_update_started ->
2      if (i == y) then
3          accept -> time_update_finished -> SKIP
4      else
5          time_update_finished -> Q(i+1,y)
```

Figura 14 - Código CSP, marcação critério ciclos temporais

Este código é equivalente ao método "ciclosTemporaisMarcacao" da Figura 10 na sessão 3.4.4 do capítulo 3. O evento `time_update_started`, inicia a representação da passagem de tempo em NAT2TEST, finalizada pelo evento `time_update_finished`.

A função espera indefinidamente pelo evento `time_update_started`, quando este ocorre, se o contador estiver marcando com valor encontrado em Y (quantidade de ciclos temporais por contraexemplo), a marca é inserida, após, espera-se indefinidamente pelo evento que representa a passagem de um ciclo. Se o contador não tiver o valor encontrado em Y, uma recursão é feita, para a espera de um novo ciclo ocorrer.

```
1   proc load { name } {
2     set sess [session]
3     $sess load [file dirname $name] [file tail $name]
4     return $sess
5   }
6
7   set qtdContraExemplos [lindex $argv 0]
8   set qtdCiclosTemporais [lindex $argv 1]
9
10  set s [load "pc.csp"]
11  set S [$s compile S -t]
12  set Z [$s compile Ze($qtdCiclosTemporais) -t]
13
14  set init "exChoice1(S,exChoice2({"
15  set final "}))"
16  set hyp [$Z refines $S]
17  set assert [$hyp assert]
18  set traces ""
19
20  while {(($assert == "false" || $assert == "xfalse") && $qtdContraExemplos > 0)} {
21
22      set dbg [$hyp debug]
23      set beh [$dbg attribution 0 1]
24      set description [$beh describe]
25
26      set result [string map {"\f" " " "Performs\t" "" "\v" ""} $description]
27      set result [string map {"_tau " "" " " ","} $result]
28      append traces $result
29      puts stdout [string map {"<" "" ">" ""} $result]
30      set resp $init$traces$final
31      append traces ","
32
33      set T [$s compile $resp -t]
34      set hyp [$Z refines $T]
35      set assert [$hyp assert]
36
37      incr qtdContraExemplos -1
38  }
```

Figura 15 - Código TCL, critério ciclos temporais

Este código é representado de forma genérica na Figura 11 do capítulo 3. Primeiramente é carregado o arquivo CSP base, que já possui dentro dele os processos marcados (linha 10). Dois processos principais são carregados, “S” (linha 11) que representa o código original do sistema a ser descrito, e “Ze” (linha 12), função que internamente chama o processo marcado pela função Q da Figura 14 deste apêndice.

Os processos S e Ze viram objetos ISM que serão transformados em objetos Hyphotesis de refinamento por traces, dos quais serão obtidos detalhes do porque da falha do refinamento, através de contraexemplos.

Com o contraexemplo em mãos, e alguns tratamentos de formato de texto, o laço continua através do refinamento entre o processo marcado e a escolha externa entre o processo original e os contraexemplos gerados.