REPRESENTAÇÃO EM CSP# DE SISTEMAS REATIVOS BASEADOS EM FLUXO DE DADOS

Trabalho de Conclusão de Curso
Engenharia da Computação

Tainã Maria dos Santos
Orientador: Prof. M.Sc. Gustavo H. P. Carvalho

UNIVERSIDADE
DE PERNAMBUCO

Tainã Maria dos Santos

REPRESENTAÇÃO EM CSP# DE SISTEMAS REATIVOS BASEADOS EM FLUXO DE DADOS

Monografia apresentada como requisito parcial para obtenção do diploma de Bacharel em Engenharia de Computação pela Escola Politécnica de Pernambuco - Universidade de Pernambuco

Universidade de Pernambuco

Escola Politécnica de Pernambuco

Graduação em Engenharia de Computação

Orientador: Prof. M.Sc. Gustavo H. P. Carvalho

Recife - PE, Brasil

24 de Julho de 2015

Declaro que revisei o Trabalho de Conclusão de Curso sob o título *"REPRESENTAÇÃO EM CSP# DE SISTEMAS REATIVOS BASEADOS EM FLUXO DE DADOS"*, de autoria de *Tainã Maria dos Santos*, e que estou de acordo com a entrega do mesmo.

Recife, _____ / _____ / _____

_____

Prof. M.Sc. Gustavo H. P. Carvalho
Orientador

A todos que me ajudaram na estrada até agora

Agradecimentos

Primeiramente, aos meus pais, pelo apoio, incentivo e paciência por 22 anos de suas vidas. Sei que sem a dedicação e abdicação de tantas coisas em suas vidas, eu não teria conseguido realizar tantos sonhos até agora. À minha irmã Thais, mesmo com todas brigas, também me apoiou em muitas de minhas decisões.

Aos amigos de faculdade Carlos, John Kennedy e Matheus, por fazerem a graduação ser algo menos triste e sombrio. Mesmo virando noites, no desespero de entregar projetos a tempo, sem as risadas e conversas descontraídas, talvez eu não conseguisse aguentar tudo isso por tanto tempo. Depois dessa experiência, sei que nossa amizade será algo longo e duradouro, muito além dos muros da Poli.

Ao meu amigo de trabalho, e também escritor, Lucas Moreno, por ter ouvido todos os meus dramas e por ter insistido para que eu terminasse logo esse trabalho para tomarmos chocolate quente com chantilly e jogar videogame por horas. Também ao meu amigo Bruno Almeida, por ter me aguentado muito mais do que pode ser considerado saudável e ter me incentivado em tantos momentos, sempre com seus discursos para nunca desistir.

A todos que me ajudaram a chegar até aqui. Sejam os professores que tive desde a infância até a graduação, sejam funcionários dos tempos de escola até os da Poli que ajudaram de alguma forma, sejam também as pessoas do meu trabalho, que tanto fizeram ajustes parar dar conta das coisas durante as minhas ausências.

Em especial, meu mais sincero agradecimento ao professor Gustavo Carvalho. Ele não é apenas um professor sensacional, mas um orientador sensacional, que dedicou sua paciência e um pouco de seu tão curto tempo para fazer com que este trabalho pudesse ser algo bom, algo que pudesse me deixar orgulhosa. Com certeza, todo o tempo em que trabalhamos juntos foi um dos mais desafiadores e mais motivadores de toda a minha graduação.

E é aqui que me despeço. Não só desse trabalho, mas também de um ciclo de cinco anos e meio que foi a minha graduação. É com alegria que agora sairei daqui para navegar em mares desconhecidos e aprender tudo o que estiver além até o último suspiro.

Resumo

A busca por qualidade em software motiva a realização de testes durante o desenvolvimento. Porém, a criação de cenários de teste está sujeita a erros e ambiguidades. Com o intuito de minimizar problemas durante a criação de casos de teste, métodos formais podem ser utilizados durante este processo de criação. Neste sentido, foi proposta a estratégia NAT2TEST, que pode representar formalmente sistemas em uma estrutura chamada DFRS, este, posteriormente, sendo representado em CSP, no dialeto CSP_M. Contudo, a representação em CSP_M limita-se a sistemas de menor complexidade em função do problema de explosão de estados. Portanto, este trabalho propõe o uso de CSP# para representação de um DFRS. Para tanto, é proposto também um algoritmo capaz de gerar a representação de DFRSs em CSP#. Também foram realizadas análises para averiguar qual ferramenta e qual dialeto (CSP_M e CSP#) teria o melhor desempenho na verificação de propriedades clássicas de CSP. Considerando as análises realizadas, mas também as limitações associadas, pode-se dizer que a representação em CSP# possui menor quantidade de estados e, portanto, permitiu a realização de análises em menos tempo.

Palavras-chave: CSP, CSP_M, CSP#, DFRS.

Abstract

The search for quality in software motivates testing during software development. However, test cases are subject to errors and ambiguities. In order to minimize the problems when creating test cases, formal methods can be used during the creation process. In this sense, it was proposed the NAT2TEST strategy, which represents systems in a formal notation called DFRS. To generate test cases, this notation is encoded as CSP_M processes. The complexity of systems that can be dealt with by this strategy is limited due to the state explosion problem – a consequence of how FDR expands and analyses CSP_M processes. Therefore, this work proposes the use of CSP# to represent DFRSs. It is also proposed an algorithm to generate a representation of DFRSs in CSP#. Futhermore, an empirical analysis was performed to determine which tool and which dialect (CSP_M and CSP#) has the best performance concerning the analysis of classical properties of CSP. Considering the analyses that were made, it can be said that the CSP# representation has few states and, therefore, carried out the analysis in less time.
Keywords: CSP, CSP_M, CSP#, DFRS. .

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

CSP	*Communicating Sequential Processes*
CSP_M	*Machine-readable Communicating Sequential Processes*
FDR	*Failures-Divergences Refinement*
PAT	*Process Analysis Toolkit*
NUS	*National University of Singapore*
DFRS	*Data-flow Reactive System*
VM	*Vending Machine*
NPP	*Nuclear Power Plant*
TIS	*Turn Indicator System*
SMT	*Satisfiability Model Theories*
LTL	*Linear Temporal Logic*
RETNA	*REquirements to Testing in a NAtural way*

Sumário

1 Introdução

Nos dias atuais, há uma dependência grande de software, o que faz com que a busca por qualidade seja um importante aspecto do desenvolvimento de software, já que o mau funcionamento do produto final pode trazer prejuízos significativos. Para minimizar os riscos, a aplicação de teste de software se faz necessária para garantir a qualidade do produto final.

Teste é útil, já que permite encontrar e corrigir erros antes que o software chegue nas mãos do usuário final. Contudo a própria criação dos testes está sujeita a erros; às vezes por definições ambíguas de requisitos, que podem levar a diferentes interpretações por parte do desenvolvedor e do cliente; às vezes por falha humana no processo de criação dos testes.

Este capítulo tem como finalidade apresentar o problema de pesquisa deste trabalho, assim como os objetivos e resultados esperados. Por fim, apresenta-se uma visão geral da estrutura deste texto.

1.1 Problema de pesquisa

Testar o funcionamento de sistemas complexos é ainda mais difícil em função da complexidade e criticidade inerentes a estes sistemas (ALMEIDA et al., 2011). Nestes casos, às vezes é necessário ter um processo de desenvolvimento apoiado por métodos formais – técnicas matemáticas utilizadas para conferir maior rigor ao desenvolvimento de um sistema (WOODCOCK et al., 2009).

Em particular, é possível usar métodos formais para a geração automática de testes. Entretanto, na prática, a necessidade de conhecimento especializado pode inviabilizar o uso dessas técnicas formais. Por essa razão, foi desenvolvida a estratégia NAT2TEST, que gera automaticamente casos de teste a partir de linguagem natural controlada (CARVALHO et al., 2014b). Essa estratégia recebe como entrada requisitos em linguagem natural e pode gerar vários formalismos a fim de representar um sistema.

Esta estratégia representa formalmente o comportamento de sistemas utilizando o formalismo chamado DFRS (*Data-Flow Reactive System*) (CARVALHO et al., 2014a). Esse DFRS trata de uso de uma representação formal para o comportamento do sistema. Posteriormente, este formalismo é representado utilizando CSP - *Communicating Sequential Processes*, no dialeto CSP_M (CARVALHO et al., 2014a), para fins de geração de testes.

Porém, o problema enfrentado ao gerar testes a partir da representação em CSP_M é a explosão de estados, tornando difícil a verificação de modelos grandes e complexos

(GOMES, 1997). Em particular, a explosão de estados é motivada pela ausência em CSP_M de um mecanismo natural de comunicação por memória compartilhada, algo que é naturalmente permitido em CSP#, outro dialeto de CSP. Portanto, a fim de obter uma maior capacidade de verificação, este trabalho aborda geração de código CSP# no contexto da estratégia NAT2TEST.

1.2 Objetivos

Este trabalho tem como objetivo a criação de um algoritmo computacional que gere código CSP# para representar o comportamento de sistemas reativos baseados em fluxo de dados, ou seja, o modelo DFRS. Com essa representação será possível o uso de alguns elementos presentes em CSP#, como a comunicação por memória compartilhada, o que visa minimizar a explosão de estados que pode ocorrer em representações de sistemas maiores em CSP_M.

O projeto também pretende analisar e comparar empiricamente o desempenho do código CSP# gerado com o código CSP_M gerado através da estratégia NAT2TEST. Essa análise visa mostrar qual representação seria mais concisa e, assim, mais apropriada à geração de casos de teste para sistemas mais complexos.

1.3 Resultados e impactos esperados

Ao final deste trabalho, espera-se concluir com sucesso a implementação de um gerador de especificações CSP# sintaticamente corretas, que descrevam o comportamento de sistemas reativos baseados em fluxos de dados.

Também se espera que este gerador permita analisar sistemas mais complexos, consumindo menos recurso computacional ao gerar casos de teste a partir da ferramenta NAT2TEST. No entanto, é importante frisar que a geração de testes a partir de especificações CSP não faz parte do escopo desta monografia.

1.4 Estrutura da monografia

Esta monografia está estruturada de acordo com os seguintes capítulos:

- Capítulo 1: qualifica o problema de pesquisa deste trabalho, assim como os objetivos e resultados esperados;
- Capítulo 2: descreve os conceitos a serem utilizados nesse trabalho, como NAT2TEST e os dialetos de CSP;

- Capítulo 3: descreve detalhadamente a representação em CSP# de um sistema reativo de fluxo baseado em dados, além de fazer um breve comparativo entre as representações nos dialetos CSP# e CSP_M;
- Capítulo 4: apresenta a análise empírica que compara o desempenho da representação em CSP# com a em CSP_M;
- Capítulo 5: apresenta as conclusões deste trabalho e possíveis trabalhos futuros.

2 Referencial Teórico

Este capítulo apresenta uma visão dos conceitos usados neste trabalho, assim como analisa trabalhos relacionados.

2.1 A estratégia NAT2TEST

NAT2TEST é uma estratégia que trabalha com geração de casos de teste a partir de de requisitos descritos em linguagem natural controlada (CARVALHO et al., 2014b). Estes requisitos descrevem o comportamento de sistemas reativos baseados em fluxo de dados. Em outras palavras, sistemas cujo comportamento pode ser descrito como reações a dados recebidos como entrada. A reação do sistema é caracterizada, então, pela produção de dados de saída.

Na figura 1 é possível ter uma visão geral do funcionamento doa estratégia NAT2TEST. Entre as etapas para a geração de casos de teste podem ser destacadas três momentos: análise sintática, análise semântica e a geração do formalismo DFRS. Após esses processos iniciais, também é possível representar esse sistema em diversos formalismos para geração de casos de teste. Entre eles, o formalismo em CSP_M para geração de teste.

De acordo com Carvalho et al. (2014b), o processo para iniciar a geração desses casos de teste se dá com uma análise sintática para a geração de árvores sintáticas. Nesta primeira fase, a estratégia preocupa-se em verificar se os requisitos do sistema estão escritos de acordo com a linguagem natural controlada definida pela estratégia. O requisito consiste na divisão de condições e ações. A condição trata das situações necessárias para que haja mudança no sistema, enquanto que a ação trata das mudanças que o sistema faz após suas variáveis atingirem os valores descritos na condição.

Na segunda etapa, as árvores sintáticas geradas pela primeira etapa são utilizadas para identificar papéis temáticos. Estes associam a cada palavra, ou a um conjunto de palavras, dos requisitos um determinado papel. Esta ideia baseia-se na teoria de Gramáticas de Casos (FILLMORE, 1967). Exemplos de papéis temáticos são: agente, a entidade responsável por realizar uma determinada ação; paciente, a entidade que sofre o resultado de uma ação, entre outros. O mapeamento desses papéis é definido através de um conjunto de regras de inferência, que, ao passar por cada nó da árvore sintática, procura por padrões que foram definidos previamente.

Feita esta análise, a ferramenta formaliza os requisitos e seus papéis temáticos através de uma estrutura chamada DFRS (*Data-Flow Reactive System*); em português, sistemas reativos baseados em fluxos de dados. Portanto, esta estrutura formaliza a

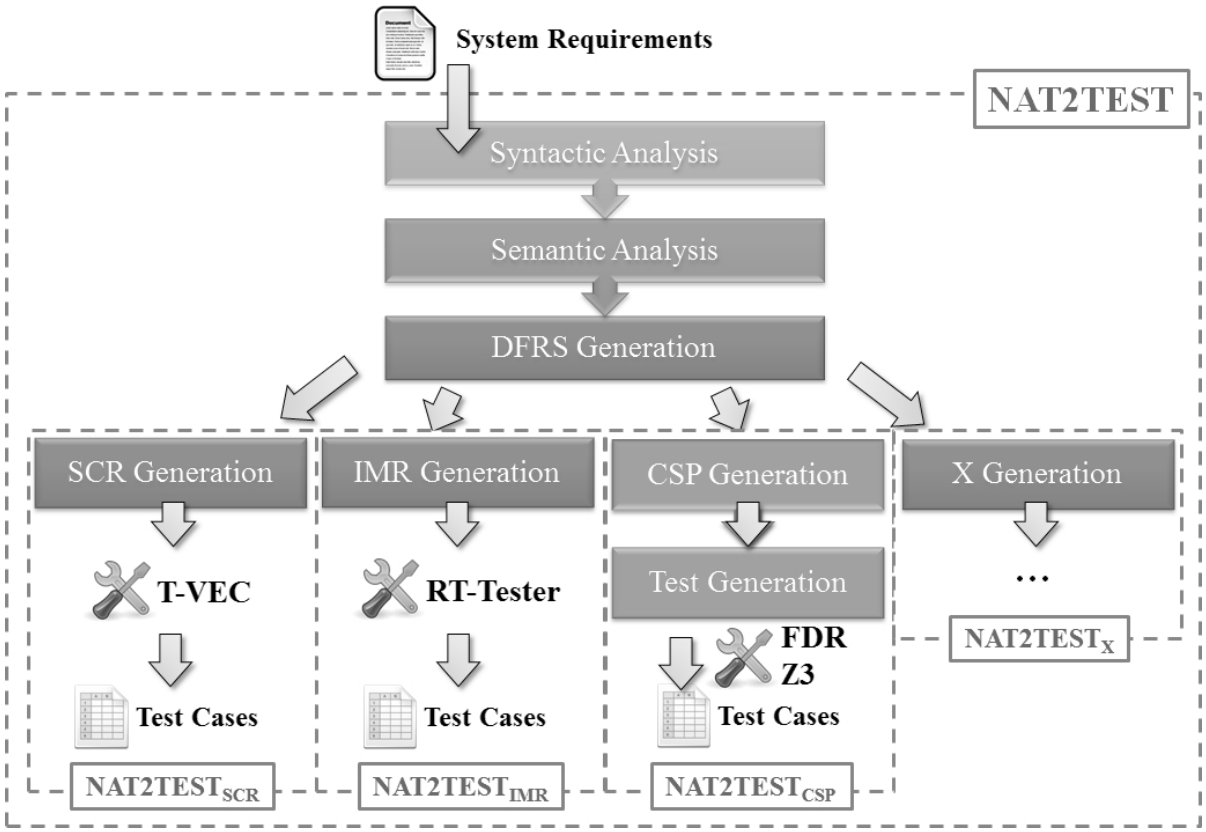

Figura 1 – Visão geral da estratégia NAT2TEST
[Fonte: (CARVALHO et al., 2014b)]

semântica informal dos requisitos.

2.2 *Data-Flow Reactive Systems*

Um DFRS pode ser definido como um sistema de transição variando de acordo com o tempo (CARVALHO et al., 2014a). Para ilustrar melhor sobre esse conceito, será usado o exemplo *Vending Machine* (VM), adaptado de Larsen, Mikucionis e Nielsen (2005), que consiste no usuário pôr uma moeda e apertar um botão para retirar o café produzido. No entanto, o tempo que o usuário leva para pressionar o botão pode resultar em dois tipos de café: fraco e forte. Se a requisição de café for feita em até 30 unidades de tempo, o café produzido será fraco, caso contrário, o café será forte. Após saber qual tipo de café produzir, a máquina leva de 10 a 30 unidades de tempo para para produzir café fraco, enquanto é preciso de 30 a 50 segundos para produzir café forte. Após a produção de café, a VM volta para o estado inicial, onde a máquina espera a inserção de uma nova moeda.

Ainda segundo Carvalho et al. (2014a), formalmente, o DFRS é formado por uma 7-tupla que contém os seguintes elementos:

- I: entradas;

- O: saídas;
- T: temporizadores;
- gcvar: relógio global;
- S: estados do sistema;
- s_0: o estado inicial do sistema;
- TR: uma relação de transição entre estados do sistema.

Os elementos de entrada e saída são compostos pelas variáveis que o sistema possui, exceto pelo relógio global (*gcvar*), usado para definir o tempo global do sistema. Cada variável tem um tipo, que pode ser inteiro, *float* ou booleano. No exemplo da VM, é possível identificar duas entradas (*the_coin_sensor e the_coffee_request_button*) e duas saídas (*the_system_mode e the_coffee_machine_output*).

Os temporizadores são elementos utilizados para condicionar o comportamento do sistema à passagem de tempo. Contudo, diferente dos elementos de entrada e saída, que são obrigatórios, os temporizadores não são necessariamente obrigatórios, podendo haver casos em que a passagem de tempo não importa para o DFRS. Nestes casos, tem-se um sistema cujo comportamento não depende da passagem de tempo. Na VM há apenas um temporizador chamado *the_request_timer*.

Um estado pode ser definido como uma relação entre as variáveis do sistema e os possíveis valores que elas podem assumir. Cada variável possui também um valor inicial, e essas relações podem ser vistas na Tabela 1. Portanto, um estado é caracterizado pelos valores assumidos por todas as variáveis do sistema em um determinado momento.

Por fim, o último elemento da tupla se trata das transições do sistema. Essas transições podem ser classificadas em funcionais e de atraso. As transições funcionais representam a reação do sistema em uma dada situação. **Exemplo:** ir para o modo *choice* e reiniciar o temporizador *the_request_timer*, quando a moeda é inserida.

As transições de atraso são caracterizadas pela passagem de tempo seguida pelo recebimento de novas entradas. **Exemplo:** o sistema se inicia no tempo zero e, após três unidades de tempo, a moeda é inserida na máquina.

Após a geração do DFRS, a estratégia NAT2TEST ainda permite que este modelo seja modelado com a álgebra de processos CSP (*Communicating Sequential Process*). A codificação em CSP de modelos DFRS permite o reuso de ferramentas disponíveis para CSP, como FDR (FORMALSYSTEMS, 1986-2010) e PAT (SUN; LIU; DONG, 2009); evitando, assim, a criação de ferramentas específicas para o modelo DFRS.

Tabela 1 – Relação entre as variáveis e seus valores iniciais do DFRS
[Fonte: Elaboração própria]

Entradas		
Variável	Tipo	Valor inicial
the_coin_sensor	booleano	*false*
the_coffee_request_button	booleano	*false*
Saídas		
Variável	Tipo	Valor inicial
the_system_mode	inteiro	*idle*
the_coffee_machine_output	inteiro	*strong*
Temporizadores		
Variável	Tipo	Valor inicial
the_request_timer	*float*	0.0

2.3 *Communicating Sequential Processes*

Segundo Roscoe (ROSCOE, 1998), CSP é uma notação matemática que descreve a interação entre os sistemas concorrentes. Existem dois dialetos de CSP: CSP_M e CSP#; descritos brevemente nas próximas seções.

2.3.1 CSP_M

CSP_M foi desenvolvido em 1998 por Bryan Scattergood e pode ser definido como a combinação da álgebra de processos CSP com linguagem uma funcional de expressões (ROSCOE, 1998).

No contexto do NAT2TEST, a notação é utilizada como uma forma de representar o DFRS. A geração do código CSP_M tem como propósito analisar o comportamento do sistema e utilizar técnicas de refinamento com FDR e resolução por SMT (*Satisfiability Module Theories*) com Z3 para a geração de casos de teste (CARVALHO; SAMPAIO; MOTA, 2013).

No Exemplo 2.1 é possível ver uma especificação em CSP_M, onde pode ser visto algumas das principais construções deste dialeto utilizadas para o entendimento deste trabalho.

Exemplo 2.1 – Exemplo em CSP_M

```
1 P = a → SKIP
2 Q = b → STOP
3 R = P [ ] Q
4 S = P ; Q
5 T = P ||| Q
6 X = P [| A |] Q
7 Y = P \backslash A
```

- **a** e **b** são chamados de eventos, estruturas atômicas que podem ser executadas ou sofrer alteração por meio de um processo.

- **P**, **Q**, **R**, **S**, **T**, **X** e **Y** são chamados de processos e representam um conjunto de eventos para compor o comportamento do sistema. Estas estruturas devem ser finalizada com um outro processo ou algum processo primitivo, como SKIP e STOP.
- **SKIP** também é um processo, porém, ele é o processo mais primitivo de CSP, ou seja, não pode ser dividido em eventos e processos menores. O SKIP indica o término com sucesso de um processo.
- **STOP**, assim como SKIP, é também um processo primitivo de CSP, porém, diferentemente do anterior, este indica o término mal-sucedido de um processo, que também pode ser chamado de *deadlock*.
- Nos processos P e Q, temos o operador chamado **prefixo**, representado por $\rightarrow$, onde os processos P e Q se comportam como o a e b, respectivamente, e, em seguida, passam a se comportar como o processo à direita.
- No processo R, tem-se uma **escolha externa** entre os processos P e Q, onde apenas um processo entre P e Q será executado. A escolha é chamada de externa pois não depende do processo R.
- No processo S, têm-se dois processos em sequência, onde o processo P executa primeiro e depois é a vez do processo Q.
- No processo T tem-se o operador de *intearleaving*, também chamado de concorrência assíncrona. Os processos P e Q são executados de maneira independente entre si, não havendo interação em qualquer evento.
- No processo X, os processos P e Q sincronizam em um conjunto de eventos chamado A, ou seja, os eventos contidos em A precisam ser realizados simultaneamente. Os demais eventos de P e Q que não estão em A ocorrem de forma assíncrona.
- O processo Y realiza o processo P onde os eventos que pertencem ao conjunto A são removidos da interface do processo. Porém, vale ressaltar que esses eventos pertencentes a A são conhecidos e descritos no processo P.

CSP_M não possui nativamente o modelo de comunicação por memória compartilhada, que se trata de uma representação explícita do conceito de variáveis globais, utilizadas em linguagens de programação mais tradicionais. Este modelo é o mais adequado para representar o DFRS, uma vez que exista o conceito de variáveis, essas variáveis podem ser alteradas de forma concorrente ou sequencial. Apesar de CSP_M permitir a representação de um modelo de memória compartilhada através de processos, esta representação não é ideal. Ao analisar estes processos, eles são primeiramente expandidos considerando todos os valores possíveis, o que pode ser custoso em função do comportamento do sistema. Esta

característica faz com que a explosão de estados seja um problema ao analisar sistemas com um maior conjunto de variáveis. Diferentemente, CSP# possui um modelo nativo de comunicação por memória compartilhada. Este modelo é tratado de forma diferente durante análises, pois considera somente os valores que as variáveis globais podem de fato assumir.

2.3.2 CSP#

Mesmo com a criação de CSP_M, diversos grupos ainda continuam a estudar outros mecanismos de mecanizar CSP. Neste meio, surgiu CSP#, idealizado pelo grupo de pesquisadores do NUS (*National University of Singapore*). Este dialeto é inspirado em linguagens de alto nível, e conta com o apoio ferramental de PAT (*Process Analysis Toolkit*) (SUN; LIU; DONG, 2009). Além da verificação de refinamentos, visualização e simulação de processos, PAT também permite a verificação de propriedades em LTL (*Linear Temporal Logic*).

A sintaxe de CSP# espelha-se muito em CSP_M, como a sincronização de eventos e construção de processos, mas vai além com recursos adicionais como comunicação através canais assíncronos (SHI et al., 2012). Para o entendimento deste trabalho, é necessário observar que, apesar de ambos os dialetos terem como base CSP, eles possuem particularidades que precisam ser pontuadas. No exemplo a seguir é possível ver o mesmo exemplo da seção anterior adaptado para a sintaxe de CSP#. As principais diferenças de sintaxe entre eles são:

Exemplo 2.2 – Exemplo em CSP#

```
1 P = a → Skip;
2 Q = b → Stop;
3 R = P [*] Q;
4 S = P ; Q;
5 T = P ||| Q;
6 X = P || Q;
7 Y = P \ A;
8 Z = a{value = 0} → Skip;
```

- Os processos primitivos *Skip* e *Stop* não são mais escritos em caixa alta.
- O operador que indica escolha externa é substituído por [*].
- Assim como a maioria das linguagens de programação mais tradicionais, é preciso um ; (ponto-e-vírgula) ao final de cada linha de código.
- Não é mais preciso informar o conjunto de eventos em que os processos estão sincronizados (ou seja, A), sendo apenas necessário o símbolo ||.
- O processo Z mostra a modificação de variáveis globais através de eventos. Esse tipo de processo é chamado de *data operations*.

2.4 Trabalhos relacionados

Na literatura é possível identificar outras formas de usar linguagem natural controlada além da estratégia NAT2TEST. PENG (SCHWITTER, 2002) é uma ferramenta que se utiliza linguagem natural controlada para escrever requisitos não ambíguos e mais precisos, utilizando de um editor de texto próprio chamado ECOLE, que indica as restrições da gramática enquanto o usuário escreve seus requisitos. Essas especificações podem ser traduzidas para estruturas de lógica de primeira ordem. Contudo, segundo Schwitter (2002), não há propostas para geração de testes de software a partir de sua ferramenta.

Como a proposta da estratégia NAT2TEST é a criação de casos de teste, é possível mencionar também a ferramenta RETNA (*REquirements to Testing in a NAtural way*) (BODDU et al., 2004), que lê requisitos escritos em linguagem natural, além de classificá-los e interagir com o usuário para refinamento desses requisitos. A ferramenta também traduz esses requisitos em lógica de primeira ordem para validação e geração de casos de teste. Além disso, RETNA tem a capacidade de lidar com sinônimos para a análise de sua gramática.

Outra forma de gerar de casos de teste pode ser vista na estratégia proposta em Nogueira, Sampaio e Mota (2014). Porém diferentemente das ferramentas mostradas anteriormente, essa estratégia utiliza CSP_M para representar formalmente o comportamento de sistemas, e a geração de teste é feita a partir de contra-exemplos, utilizando o processo de refinamento de modelos com a ferramenta FDR. A vantagem deste processo consiste na facilidade de gerar de casos de teste a partir desses modelos, sem que se faça necessário o desenvolvimento de algoritmos para o processo de geração.

Apesar de não existir uma ferramenta que possa traduzir essa representação gerada em CSP_M para CSP# a fim de obter casos de teste, a proposta para este trabalho é obter a representação em CSP# a partir do DFRS gerado através da ferramenta NAT2TEST.

Uma das principais vantagens de se usar CSP# para essa representação é o uso de memória compartilhada, através de variáveis globais, não sendo necessária a criação de processos para que haja tal simulação. Contudo, em CSP#, o usuário precisa especificar o alfabeto dos processos criados, onde nele estará contido o conjunto de eventos que compõem o processo, com os possíveis valores que esses eventos podem assumir, de acordo com a lista de valores que o DFRS disponibiliza para cada variável. Outro ponto negativo é a falta de sincronização múltipla entre eventos comunicados através de canais, algo que não é necessário neste trabalho.

3 Representacão em CSP# de DFRSs

Contudo, a proposta deste trabalho é integrar à ferramenta uma forma de representar DFRSs em CSP#, com o intuito de evitar o problema da explosão de estados, antes mencionado. Para exemplificar o trabalho proposto por esta monografia, será considerado como exemplo a VM, citada anteriormente.

3.1 Visão geral

O gerador de especificações CSP# a partir de modelos DFRS foi implementado usando a linguagem Java. O propósito é formar uma representação alternativa à representação em CSP_M, apesar das limitações que CSP# possui, mesmo com o mecanismo de memória compartilhada. Apesar de não fazer parte do escopo deste trabalho analisar se a semântica do código CSP# gerado possui uma equivalência com a do código CSP_M correspondente, deve-se observar se a especificação CSP# do DFRS respeita propriedades básicas do modelo DFRS; por exemplo: se a mesma é livre de *deadlock*.

Na Figura 2, é possível visualizar a arquitetura do gerador de código CSP# para modelos DFRSs. Por conta da legibilidade da imagem, os parâmetros foram omitidos do diagrama. No pacote *cspSharpGenerator*, cada classe é responsável por uma parte do algoritmo de geração de código CSP#. A classe *CSPSharpGenerator* implementa a interface *ICSPSharpGenerator*, que contém um único método chamado *generateCSP*. Na classe *CSPSharpGenerator*, esse método fica responsável por chamar todas as classes que coletivamente representam o algoritmo em questão. Antes da geração, é necessário definir as listas de variáveis, temporizadores e *eta*s, que serão utilizadas pelas classes que formam o algoritmo. Estas informações são geradas a partir do método *generateVariablesAndEta*, onde o objeto DFRS é passado como parâmetro. Na formação do algoritmo de geração de código CSP#, temos as seguinte classes:

- *VariableGenerator*: responsável pela representação das variáveis;
- *InputsGenerator*: responsável pela interação do sistema com o ambiente;
- *SysBehaviorGenerator*: representa o comportamento do sistema;
- *DelayGenerator*: modela o tempo do sistema simbolicamente;
- *FunTransGenerator*: representa as transições funcionais do sistema;
- *OutputsGenerator*: responsável por criar o evento composto *output*;

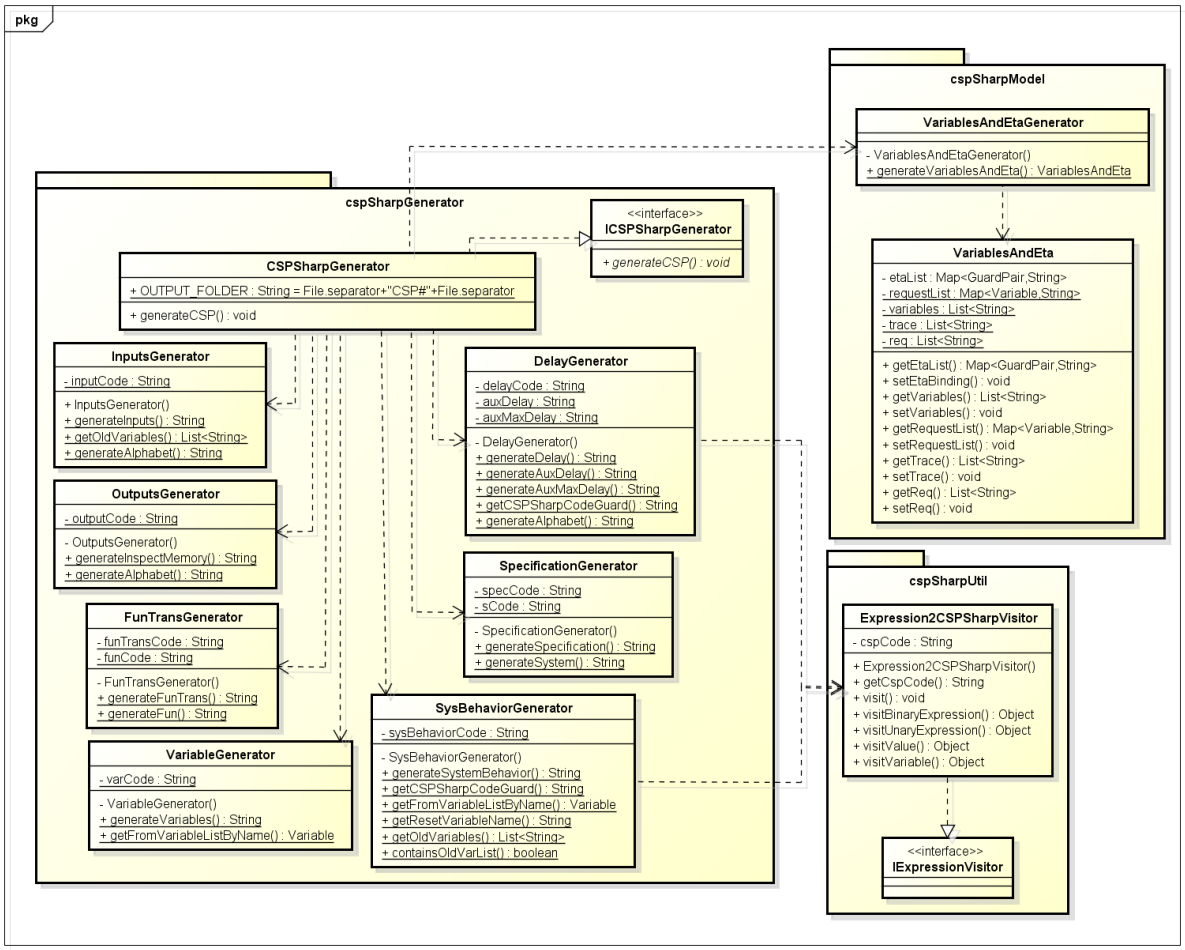

Figura 2 – Diagrama de classes do gerador de código CSP#
[Fonte: Elaboração própria]

- *SpecificationGenerator*: representa o comportamento cíclico do sistema.

Na próxima seção, cada classe citada será apresentada a fim de se obter maior entendimento no algoritmo que gera o código em CSP#.

3.2 Geração em CSP#

Esta seção apresentará o passo-a-passo necessário para gerar especificações CSP# a partir de DFRSs. Inicialmente, são identificadas as variáveis do modelo. Em seguida, são gerados os processos que descrevem a interação do sistema com o seu ambiente. Por fim, derivam-se processos para descrever o próprio comportamento do sistema, através da realização alternada de transições funcionais e de atraso.

3.2.1 Identificação de variáveis

Em CSP#, são declaradas variáveis globais para representar as entradas e saídas do sistema. Também são declaradas variáveis auxilares, utilizadas na definição dos processos que descrevem o comportamento do DFRS.

Por CSP# ter uma representação explícita de variáveis globais (memória compartilhada), não é necessária a criação de processos para simular este tipo de comunicação; no exemplo 3.1 é possível ver como era feito em CSP_M a declaração das variáveis. É possível ver neste exemplo que após o mapeamento de todos os possíveis valores que cada variável pode assumir, é feito a representação das variáveis do sistema através do *datatype* chamado VAR, enquanto o *datatype* TYPE representa os tipos de possíveis valores. No exemplo da VM é possível encontrar valores do tipo booleano (B) e inteiro (I). O processo MCELL representa uma célula de memória onde podem ser armazenados variáveis e seus valores. Para cada alteração nesses valores é necessário que cada célula seja lida e atualizada pelos canais *get* e *set*. O processo MEMORY compõe todas as células de memória em paralelo, criando assim uma representação de memória, e requer as definições do *datatype* VAR e de *binding* para mapear os valores iniciais para cada variável do sistema.

Exemplo 3.1 – Representação em CSP_M das variáveis de VM

```
1  the_system_mode_values = {0, 1, 2, 3}
2  (...)
3  range(I_the_system_mode) = {I_the_system_mode.0, ...}
4  (...)
5
6  datatype TYPE = I_the_system_mode.the_system_mode_values | ...
7  datatype VAR = funTrans | the_system_mode | ...
8  initialBinding = {(the_system_mode, I_the_system_mode.1), ...}
9
10 channel get, set: VAR.TYPE
11 MCELL(var,val) = get!var!val -> MCELL(var,val)
12 [] set!var?val' : range(tag(val)) -> MCELL(var,val')
13 MEMORY(binding) = ||| (var,val) : binding @ MCELL(var,val)
14 SYSTEM_MEMORY = MEMORY(initialBinding)
```

Já no exemplo 3.2, também exemplificando a criação de variáveis para VM , no entanto, aqui para CSP#, é possível ver a representação explícita das variáveis a partir de memória compartilhada. Cada uma dessas variáveis é declarada e inicializada com o seu respectivo valor inicial declarado no DFRS. Também não há a necessidade de informar todos os possíveis valores que essas variáveis do sistema podem assumir, diferente do que ocorre em CSP_M. Além das variáveis do sistema, também são declaradas variáveis globais necessárias à representação do comportamento do DFRS; por exemplo, *funTrans*, que indica o acontecimento de uma transição funcional no sistema.

Exemplo 3.2 – Representação em CSP# das variáveis de VM

```
1 var funTrans = false;
2 var the_coffee_request_button = false;
3 var the_coin_sensor = false;
4 var the_system_mode = 1;
```

```
5  var the_coffee_machine_output = 0;
6
7  var eta1 = false;
8  var eta2 = false;
9  var eta3 = false;
10 var eta4 = false;
11
12 var the_request_timer = false;
13
14 var old_the_coffee_request_button = false;
15 var old_the_coin_sensor = false;
```

A seguir, é possível visualizar o Algoritmo 1, responsável pela geração das entradas e saídas do sistema como variáveis globais para a representação em CSP#.

A linha 1 declara *variablesCSP*, uma *string* que constitui o retorno deste algoritmo. A linha seguinte percorre todas as variáveis definidas no DFRS. Cada variável é composta pelo seu nome, o tipo, o valor inicial e o conjunto de possíveis valores que ela pode assumir. Na linha 4, a variável *var* recebe o nome de uma variável do sistema e verifica na linha 5 se ela está contida na lista de variáveis, *varList*, onde estão armazenadas todas as variáveis do sistema. Caso esteja contida, o próximo passo é verificar o tipo da variável. A linha 6 verifica se a variável é do tipo inteiro ou booleano. Caso uma das condições seja verdadeira, a variável *variablesCSP* recebe o nome da variável e se inicializa considerando seu valor inicial declarado no DFRS (linha 7). Caso a variável seja de um tipo não suportado pela linguagem CSP, como ponto flutuante, uma exceção é mostrada (linhas 8 – 9), e o processo de representação do DFRS em CSP# interrompido. Não são criadas variáveis para representar o valor concreto dos temporizadores, uma vez que o tempo é representado de forma simbólica tanto no código CSP_M, como no CSP#. Esta representação simbólica é detalhada a seguir.

Após as declarações das variáveis de entrada e saída, o próximo passo é percorrer as funções que o DFRS possui (linha 11), ainda no Algoritmo 1, visitando as entradas de cada função, como é possível ver na linha 12. Ao verificar o mapeamento entre ($discreteGuard \times timedGuard$), caso este não seja nulo (linha 13), é criada uma variável *eta* correspondente, como pode ser visto na linha 14. A ideia por trás da criação de variáveis *eta* é abstrair a passagem de tempo. Para cada condição temporal (*timedGuard*), cria-se uma variável *eta*. Quando esta tiver o valor *true*, significa que passou algum tempo que satisfaz a respectiva guarda temporal. No entanto, em CSP não se saberá qual é o valor concreto deste tempo que passou. Este valor será posteriormente encontrado com o auxílio do Z3.

Em seguida, o próximo passo consiste em criar variáveis auxiliares para cada temporizador. Como dito anteriormente, não se modela em CSP o valor concreto destes temporizadores. No entanto, estas variáveis auxiliares são necessárias para garantir que as transições funcionais possuem efeito colateral. Ou seja, uma transição funcional do sistema só é realizada se ela modifica o valor atual de pelo menos uma variável do sistema, inclusive, dos temporizadores. Portanto, estas variáveis auxiliares serão inicializadas como *false*, e

Algoritmo 1: Variáveis

Input: dfrs, varList
Output: variablesCSP

```
1  variablesCSP = newString();
2  variablesCSP = variablesCSP + "varfunTrans = false;";
3  for var ∈ dfrs.I, dfrs.O do
4      varName = var.name;
5      if varList.name.contains(varName) then
6          if var.type = integer or var.type = boolean then
7              variablesCSP = variablesCSP + "var" + varName + " = " + var.initialValue";";
8          else
9              throwExcpetion(FloatnumbersoranothertypeofvariablearenotyetsupportedintheCSPlevel);
10 etaTime = 1;
11 for f ∈ dfrs.F do
12     for discretGuard, timedGuard, actionList ∈ f do
13         if timedGuard ≠ null then
14             etaName = "eta" + etaTime;
15             variablesCSP = variablesCSP + "var" + etaName + " = false;";
16             etaTime = etaTime + 1;
17 for reqVar ∈ dfrs.T do
18     variablesCSP = variablesCSP + "var" + reqVar.name + " = false;";
19 for variable ∈ varList do
20     if variable.contains(prev) then
21         variable.replace("prev", "old_");
22         variablesCSP = variablesCSP + "var" + variable + " = " + variable.initialValue";";
```

quando o sistema modificar seu valor, serão atualizadas para *true*. Desta forma, apesar de não saber o valor concreto de um temporizador, será possível saber se o mesmo foi ou não modificado por uma dada transição funcional. Na linha 17, a lista de temporizadores do DFRS é percorrida, e na linha 18, esses temporizadores são declarados e inicializados com o valor *false*.

Por fim, a última etapa deste processo de representar as variáveis se dá com a criação de variáveis com prefixo *old* a partir do uso de *prev*: um recurso do DFRS para referenciar o valor anterior de determinadas variáveis. O processo consiste em percorrer a lista de variáveis e checar as variáveis com este prefixo *prev*, para substitui-los pelo prefixo *old* e inicializar estas variáveis com seu respectivo valor inicial (linhas 19 – 22). O intuito de gerar essas variáveis com prefixo *old* é que algumas funções há a necessidade de ter o conhecimento do valor armazenado anteriormente para algumas variáveis em determinadas condições para que possa haver a mudança de estado do sistema. Para o exemplo da VM, em determinados momentos há a necessidade de se saber os valores anteriores das variáveis de entrada *the_coffee_request_button* e *the_coin_sensor*, declarando-as novamente com o prefixo *old*.

Ao executar esta primeira parte da geração de código CSP#, é possível obter todas as variáveis do sistema.

3.2.2 Interação com o ambiente

Após a geração das variáveis, o próximo passo é representar a interação do sistema com o ambiente, que irá simular o recebimento de novas entradas. Em CSP_M, como não há o conceito das variáveis, como explicado anteriormente, simula-se o recebimento de novas entradas a partir de valores comunicados através de canais. Como é possível ver no exemplo 3.3, cria-se um canal *c_the_coffee_request_button* a partir do qual recebe-se um valor, neste caso, *true* ou *false*. Este valor, que é armazenado em *newV_the_coffee_request_button*, é, em seguida, armazenado na respectiva célular de memória através do evento *set!the_coffee_request_button.B.newV_the_coffee_request_button*. Antes, é preciso copiar o valor atual desta variável para a posição de memória referente a versão anterior. Isto é feito pelos eventos get e set que aparecem antes do set mencionado anteriormente.

Exemplo 3.3 – Representação em CSP_M de *Inputs*

```
1 INPUTS =
2 c_the_coffee_request_button?newV_the_coffee_request_button ->
3 get!the_coffee_request_button?B.v_the_coffee_request_button ->
4 set!old_the_coffee_request_button!B.v_the_coffee_request_button ->
5 set!the_coffee_request_button!B.newV_the_coffee_request_button -> (...) ->
6 input.the_coffee_request_button.B.newV_the_coffee_request_button.(...) -> SKIP
```

Já em CSP#, cada variável tem sue valor atual passado para sua correspondente *old* através do processo de *data operation*, evento utilizado para a atualização de valores no CSP#. Após essa atualização, cada variável de entrada passará por um processo de escolha externa com o conjunto de valores que estas variáveis podem assumir, como é possível visualizar no exemplo 3.4 feito para o caso da VM.

Exemplo 3.4 – Representação em CSP# de *Inputs*

```
1  IN_the_coffee_request_button() =
2  the_coffee_request_button_old{old_the_coffee_request_button =
3  the_coffee_request_button} -> (
4  (the_coffee_request_button_false{the_coffee_request_button = false} -> Skip)
5  [*]
6  (the_coffee_request_button_true{the_coffee_request_button = true} -> Skip)
7  );
8
9  (...)
10
11 INPUTS() = IN_the_coffee_request_button();IN_the_coin_sensor();
12 input.the_coffee_request_button.the_coin_sensor -> Skip;
```

A seguir, é possível visualizar o Algoritmo 2, responsável pela geração dessas interações com o ambiente através do recebimento de novas entradas.

A linha 1 declara *inputsCSP*, uma *string* que constitui o retorno deste algoritmo. Na linha 2, o algoritmo identifica, no primeiro momento, as variáveis de entrada que possuem o prefixo *old* e armazena essas variáveis, caso esta contenha este prefixo, na variável *auxName* (linhas 5 – 7). Após este processo, o algoritmo deve identificar o tipo

de variável em questão. Na linha 9, o algoritmo irá verificar se a variável, após identificar que a variável possui o prefixo *old* (linha 8), é do tipo booleano. Caso a condição seja satisfeita no algoritmo, a próxima etapa consiste em retirar o prefixo da variável auxiliar e criar uma variável chamada *event*, onde primeiro a variável atribui seu valor atual para sua correspondente *old* para, em seguida, representar uma escolha externa com todos os valores possíveis para a variável em questão no algoritmo (linhas 10 – 16).

Caso a variável de entrada não seja do tipo booleano, o algoritmo cai na condição onde a variável é do tipo inteiro (linhas 18 – 30), onde o mesmo processo é feito, porém, agora para variáveis do tipo inteiro. O que diferencia essa parte do algoritmo é a lista de valores possíveis. Não há como saber se elas são uma *string*, onde seria necessário converter em um número natural para representá-las ou de fato um número inteiro. Para contornar este problema, é necessário que haja um try/catch para verificar se os valores são um número convertido em *string*, como pode ser visto na linha 23. Caso não seja, a linha 25 trata a exceção de forma a converter em números cada item da lista de valores possíveis da variável.

Não é preciso verificar se a variável é de qualquer outro tipo. No Algoritmo 1 é feito a validação dos tipos das variáveis e o algoritmo é interrompido caso os valores não sejam do tipo inteiro ou booleano, não se fazendo necessário uma nova verificação neste ponto.

Ao final deste processo, é verificado se *event* já não existe em uma lista chamada *eventVar* e se a variável *auxName* é diferente de nulo na linha 31. Caso a condição seja verdadeira, adiciona-se a variável *event* na lista (linha 32) e é armazenado o valor de *auxName* em uma outra lista chamada *eventName*, acompanhada do prefixo *IN_* (linha 33). O propósito para este trecho do algoritmo é para evitar processos com nomes repetidos no código CSP#.

Após esse processo ser feito com todas as variáveis de entrada, é mostrado nas linhas 34 e 35, a lista *eventVar* é percorrida para adicionarmos à variável *inputCSP* a representação das escolhas externas. Em seguida, na linha 36, adicionamos o nome do processo *INPUTS*, para que, nas linhas 37 e 38, as chamadas dos processos de escolha externa sejam acrescentadas ao processo *INPUTS*.

Por fim, criamos um evento composto chamado *input*, considerando as variáveis de entrada, seguido pelo processo *Skip* (linhas 39 – 41). Este evento composto representa as entradas recebidas pelo sistema em um dado instante de tempo.

3.2.3 Comportamento do sistema

Diferentemente das representações das variáveis e da interação com o ambiente, a maior parte das diferenças ao representar em CSP_M ou em CSP# o comportamento do sistema é puramente sintática. Em CSP_M, em função do modelo utilizado para representar

Algoritmo 2: *Inputs*

Input: dfrs, varList
Output: inputCSP

```
1  inputCSP = newString();
2  oldVar = getOldVariables(varList)
3  eventVar = newArrayList < String > ()
4  for var ∈ dfrs.I do
5      for oldVar ∈ varList do
6          if oldVar.name.contains(var.name) then
7              auxName = oldVar.name;
8      if auxName.contains("old_") then
9          if var.type == boolean then
10             auxName = auxName.replace("old_", "");
11             event = "IN_" + auxName + "() = " + auxName + "_old{old_" + auxName + " =
               "auxName + "} − > (";
12             for expValue ∈ var.expectedValues do
13                 event = event + (auxName + "_" + expValue + "{" + auxName + " =
                   " + expValue + "} − > Skip)[∗]";
14                 if var.indexOf(var.expectedValue) = var.expectedValue.size() − 1 then
15                     event = event.substring(0, event.lenght() − 6);
16                     event = event + "); ";
17         else
18             auxName = auxName.replace("old_", "");
19             event = "IN_" + auxName + "() = " + auxName + "_old{old_" + auxName + " =
               "auxName + "} − > (";
20             for expValue ∈ var.expectedValues do
21                 mode = 0
22                 try{
23                     event = event + (auxName + "_" + Integer.parseInt(expValue) + "{" +
                       auxName + " = " + Integer.parseInt(expValue) + "} − > Skip)[∗]";
24                 }catch(ExcepetionE){
25                     event = event + (auxName + "_" + mode + "{" + auxName + " =
                       " + mode + "} − > Skip)[∗]";
26                 }
27                 mode = mode + 1;
28                 if var.indexOf(var.expectedValue) = var.expectedValue.size() − 1 then
29                     event = event.substring(0, event.lenght() − 6);
30                     event = event + "); ";
31     if !eventVar.contains(event) and auxName ≠ null then
32         eventVar.add(event);
33         eventName.add("IN_" + auxName + "()");
34 for eVar ∈ eventVar do
35     inputCSP = inputCSP + eVar;
36 inputCSP = inputCSP + "INPUTS() = ";
37 for eName ∈ eventName do
38     inputCSP = inputCSP + eName + "; ";
39 inputCSP = inputCSP + "input";
40 for variable ∈ dfrs.I do
41     inputCSP = inputCSP + "." + variable.name;
42 inputCSP = inputCSP + " − > Skip; ";
```

memória compartilhada, as variáveis do sistema precisam ser carregadas e passadas como argumentos antes de serem utilizadas. No processo de *SYSTEM_BEHAVIOR*, todas as variáveis utilizadas são passadas como parâmetro do processo(por exemplo, ver o processo *SYSTEM_BEHAVIOUR* no exemplo 3.5).

Exemplo 3.5 – Representação em CSP_M do comportamento do sitema de VM

```
1  SYSTEM_BEHAVIOUR(v_the_system_mode, v_eta1, v_the_coffee_machine_output,
2  v_old_the_coffee_request_button, v_the_coffee_request_button,
3  v_old_the_coin_sensor, v_the_coin_sensor, v_eta2,
4  v_the_request_timer, v_eta3, v_eta4) =
5  ((v_the_system_mode == 2) and v_eta1 and (v_the_system_mode != 1 or
6  v_the_coffee_machine_output != 0) &
7  set!funTrans!B.true -> set!the_system_mode!I_the_system_mode.1 ->
8  set!the_coffee_machine_output!I_the_coffee_machine_output.0 -> REQ005 -> SKIP)
9  []
10 (...)
11 []
12 (not((v_the_system_mode == 2) and v_eta1 and (v_the_system_mode != 1 or
13 v_the_coffee_machine_output != 0)) (...) &
14 SKIP)
```

Em CSP#, com o conceito de variáveis globais sendo aplicado, não há a necessidade de passar valores como argumenos, já que as variáveis serão acessadas diretamente e serão atualizadas através de *data operations*. No exemplo 3.6 é possível ver parte da representação do comportamento do sistema para a VM.

Exemplo 3.6 – Representação em CSP# do comportamento do sitema de VM

```
1  SYSTEM_BEHAVIOUR() =
2  ([((((!((old_the_coffee_request_button == true)) &&
3  (the_coffee_request_button == true)) && (old_the_coin_sensor == false)) &&
4  (the_coin_sensor == false)) &&
5  (the_system_mode == 0)) && eta1 &&
6  (the_request_timer != true || the_system_mode != 2)]
7  funTrans_true{funTrans = true} ->
8  reset_the_request_timer{the_request_timer = true} ->
9  the_system_mode_2{the_system_mode = 2} ->REQ003 ->Skip)
10 [*]
11 (...)
12 [*]
13 ([!(((((!((old_the_coffee_request_button == true)) &&
14 (the_coffee_request_button == true)) && (old_the_coin_sensor == false)) &&
15 (the_coin_sensor == false)) && (the_system_mode == 0)) &&
16 eta1 && (the_request_timer != true || the_system_mode != 2)) && (...)]
17 Skip);
```

Após as duas etapas iniciais, agora é o momento de representar o comportamento do sistema de acordo com cada função que está definida no DFRS, simulando o estado do sistema após uma transição funcional, como é possível ver no Algoritmo 3.

Primeiramente, é inicializada a variável retorno e atribuímos a ela o nome do processo *SYSTEM_BEHAVIOUR* (linhas 1 – 2). Em seguida, na linha 5, para cada função do DFRS, o algoritmo irá gerar uma escolha externa considerando cada reação do sistema. Lembrando que cada reação está condicionada a certa situação, esta última descrita pelo

par ($discreteGuard \times timedGuard$). O método auxiliar *changeFormat* é responsável por transformar estas guardas para a sintaxe de CSP#.

Em seguida, é verificado se há alguma variável *eta* relacionada ao *timedGuard*, e acrescentamos à condição da guarda em uma conjunção (linha 13 – 15). Esta ação está associada à representação simbólica da evolução do tempo, como descrito anteriormente.

Para concluir a formação da guarda, observa-se que uma última condição (linhas 17 – 25) será adicionada. Esta condição garante que a reação do sistema só será realizada se a execução da mesma tiver um efeito colateral. Para isso, é necessário percorrer *actionList* e averiguar os valores que serão assumidos pelas variáveis do DFRS que sofrerão alteração, então adicioná-las à condição da guarda de forma a verificar se essas variáveis já não possuem atualmente o valor que será atribuído pelas reações do sistema. O mesmo deve ser feito também com os temporizadores, caso eles existam (linhas 22 – 24). Esta última condição garante que uma transição funcional só será realizada se a mesma possuir efeito colateral; ou seja, modificar o valor atual de pelo menos uma de suas variáveis.

Em seguida, as condições são agrupadas em uma conjunção de negações na variável *combinedNegation*. Esse processo é necessário para a compôr a última condição, quando não há nenhuma reação do sistema a ser observada, uma vez que nenhuma condição é verdadeira.

Para compor as ações associadas a cada guarda, é necessário percorrer *actionList* e representar através de eventos do tipo *data operation* as alterações sofridas pelas variáveis, e, por fim, adicionar *Skip* ao processo (linhas 33 – 44). Para cada item de *actionList* é verificado se este está contido na lista de temporizadores, como é possível ver na linha 35, e caso a condição seja satisfeita, será atribuído à variável de retorno *sysBehaviorCSP* um evento de *reset*, onde esse evento atribui *true* ao temporizador do sistema (linhas 36 – 38). Então, é criado um evento onde é atribuído o novo valor à variável que sofre alteração no sistema. Contudo, antes de criar esse evento, é verificado se a variável em questão tem uma correspondente com o prefixo *old* (linha 41). Se a condição for verdadeira, antes de criar o evento onde à variável sofrerá alteração, é criado um evento onde é passado o valor atual para a variável de prefixo *old*. Por fim, é acrescentado à ação um evento que representa o identificador do requisito do sistema, seguido pelo evento *Skip*.

Após esse processo se repetir para todas as funções do DFRS, é adicionado à variável de retorno na linha 45, *sysBehaviorCSP*, uma última escolha, formada pela variável *combinedNegation*, onde caso essa condição seja satisfeita, apenas é feito o evento *Skip*. Como dito anteriormente, esta condição descreve a situação quando o sistema não tem nenhuma reação específica, uma vez que todas as condições são falsas.

Algoritmo 3: Comportamento do Sistema

Input: dfrs, varList, etaBingind, requestList

Output: sysBehaviorCSP

```
1  sysBehaviorCSP = newString();
2  sysBehaviorCSP = "SYSTEM_BEHAVIOR() = ";
3  firstCondition = true;
4  oldVarList = getOldVariables(varList);
5  for componnt ∈ dfrs.F.getKeys() do
6      function = dfrs.F.get(component);
7      for (discretGuard, timedGuard, actionList) ∈ function do
8          if firstCondition then
9              firstCondition = false;
10         else
11             sysBehaviorCSP = sysBehaviorCSP + "[*]";
12         guardCode = changeFormat(discretGuard);
13         if timedGuard ≠ null then
14             etaName = etaBinding.find((discretGuard, timedGuard));
15             guardCode = guardCode + "&&" + etaName;
16         guardCode = guardCode + "&&(";
17         for action ∈ actionList do
18             varName = action.variable;
19             value = changeFormat(action.expression);
20             if dfrs.T.find(varName) = null then
21                 guardCode = guardCode + varName + "! = " + value;
22             else
23                 variable = dfrs.T.find(varName);
24                 guardCode = guardCode + requestList.variable + "! = true";
25             guardCode = guardCode + "||";
26         guardCode = guardCode.subString(0, guardCode.lenght() − 4);
27         guardCode = guardCode + ")";
28         if combinedNegation = null then
29             combinedNegation = "!(" + guardCode + ")";
30         else
31             combinedNegation = combinedNegation + "&&!(" + guardCode + ")";
32         sysBehaviorCSP = sysBehaviorCSP + "([" + guardCode
           + "]funTrans_true{funTrans = true} − > ";
33         for action ∈ actionList do
34             varName = action.variable;
35             if dfrs.T.find(varName) ≠ null then
36                 variable = dfrs.T.get(varName);
37                 resetName = requestList.get(variable);
38                 sysBehaviorCSP =
                   sysBehaviorCSP + "reset_" + resetName + "{" + resetName + " = true} − > ";
39             else
40                 value = changeFormat(action.expression);
41                 if containsOldVarList(oldVarList, varName) then
42                     sysBehaviorCSP = sysBehaviorCSP + "old_" + varName + "_" + value +
                       "{old_" + varName + " = " + varName + "} − >
                       " + varName + "_" + value + "{" + varName + " = " + value + "} − > ";
43                 else
44                     sysBehaviorCSP =
                       sysBehaviorCSP+varName+"_"+value+"{"+varName+" = "+value+"} − > ";
45         sysBehaviorCSP = sysBehaviorCSP + discretGuard.id + " − > Skip)";
46 sysBehaviorCSP = sysBehaviorCSP + "[*]([" + combinedNegation + "]Skip); ";
```

3.2.4 Modelando o tempo simbolicamente

Após representar o comportamento do sistema, é preciso também representar a evolução do tempo, conforme descrito em Carvalho, Sampaio e Mota (2013). Para cada função do DFRS existe guardas temporais associadas a cada função do sistema. Caso a guarda temporal para determinada função seja diferente de nulo, essa guarda terá uma variável *eta* associada, como explicado anteriormente. Vale salientar novamente que nas representações em CSP (tanto no CSP_M quanto no CSP#) não há o conhecimento de quanto tempo se passou no momento, por conta da limitação da própria linguagem. O intuito desta modelagem é informar se houve a passagem do tempo entre as funções do sistema.

Para as representações em CSP_M vale salientar que inicialmente, assim como ocorre com a interação com o ambiente, é necessário carregar todas as variáveis do sistema utilizando a leitura através de canais *get*. Além disso, ainda há a necessidade de alterar os valores das variáveis *eta* para o valor *false*, e para isso é utilizado o canal *set* para a atualização dessas variáveis. Já em CSP#, a atualização é feita direto nas variáveis globais com o uso dos eventos de *data operation*. Além dessas diferenças, também há a diferença sintática entre ambos os dialetos, como é possível ver nos exemplos 3.7, para CSP_M, e 3.8, para CSP#, ambos considerando o caso da VM.

Exemplo 3.7 – Representação em CSP_M da modelagem simbólica do tempo de VM

```
DELAY(delayChannel) =
set!eta4!B.false ->
set!eta3!B.false ->
set!eta2!B.false ->
set!eta1!B.false ->
get!the_system_mode?I_the_system_mode.v_the_system_mode ->
get!old_the_coffee_request_button?B.v_old_the_coffee_request_button ->
get!the_coffee_request_button?B.v_the_coffee_request_button ->
get!old_the_coin_sensor?B.v_old_the_coin_sensor ->
get!the_coin_sensor?B.v_the_coin_sensor ->
(
((v_the_system_mode == 2) & set!eta1!B.true -> SKIP)
[]
(...)
[]
(not((v_the_system_mode == 2)) and (...) & SKIP)
)
; get!eta1?v_eta1 -> get!eta2?v_eta2 -> get!eta3?v_eta3 -> get!eta4?v_eta4 ->
delayChannel.eta1.v_eta1.eta2.v_eta2.eta3.v_eta3.eta4.v_eta4 -> SKIP
```

Exemplo 3.8 – Representação em CSP# da modelagem simbólica do tempo de VM

```
DELAY(isMaxDelay) =
eta1_false{eta1 = false} -> eta2_false{eta2 = false} ->
eta3_false{eta3 = false} -> eta4_false{eta4 = false} -> Skip;
(([[[[[!((old_the_coffee_request_button == true)) &&
(the_coffee_request_button == true)) && (old_the_coin_sensor == false)) &&
(the_coin_sensor == false)) && (the_system_mode == 0))]
eta1_true{eta1 = true} -> Skip)
[*]
(...)
```

```
10 [*]
11 ([!(((((!((old_the_coffee_request_button == true)) &&
12 (the_coffee_request_button == true)) && (old_the_coin_sensor == false)) &&
13 (the_coin_sensor == false)) && (the_system_mode == 0))) && (...)]
14 zeta -> Skip)
15 ); if(isMaxDelay){ AUX_MAXDELAY() } else { AUX_DELAY() };
```

Para o exemplo da VM, existem quatro situações onde a quantidade de tempo decorrido interfere no comportamento do sistema: (1) se o usuário fez a requisição de café em até 30 unidades de tempo após ter inserido a moeda, (2) se esta requisição ocorreu em mais de 30 unidades de tempo, (3) se já passou o tempo necessário para produzir café fraco, e (4) se já passou o tempo necessário para produzir café forte. Portanto, quatro variáveis *eta* são criadas para representar estas quatro situações. Ao atribuir *true* para uma destas variáveis, assume-se que o tempo decorrido satisfazer a respectiva condição temporal. O Algoritmo 4 simula o estado do sistema em que se verifica se a quantidade de tempo que passa interfere no comportamento do sistema.

O primeiro passo é inicializar a variável de retorno *delayCSP*, na linha 1, antes de criar as escolhas externas entre as guardas cujo *timedGuard* não seja nulo. Assim como foi feito na seção anterior, o algoritmo percorrerá todas as funções do DFRS, onde, para cada guarda se tem a relação ($discreteGuard \times timedGuard$) (linhas 2 – 3). Em cada relação, o algoritmo irá verificar se *timedGuard* é diferente de nulo. Sendo este o caso, será necessário inicializar com *false* a variável *eta* correspondente. Esta inicialização é salva na *string* $etaSetFalse$ (linhas 4 – 6). Para o caso da variável *etaSetFalse* ser nula, no lugar das atribuições no início do processo *DELAY*, se tem apenas o processo *Skip*;

Em seguida, após as inicializações, todas as funções serão percorridas mais uma vez pelo algoritmo, onde é verificado para cada elemento ($discreteGuard \times timedGuard \rightarrow actionList$) se *timedGuard* é, mais uma vez, diferente de nulo (linhas 11 – 13). Caso a condição seja satisfeita, tem-se o mesmo processo mostrado no Algoritmo 3, onde, para cada guarda, o valor da variável *eta* correspondente será *true*, enquanto que as condições das guardas serão adicionadas à *combinedNegation* e agrupadas em uma conjunção de negações (linha 14 – 24).

Todo este código, cuja geração foi explicada no parágrafo anterior, é salvo na variável *delayCondition*. Caso esta variável seja igual a nulo, não existem *eta*s para esse sistema e será atribuído para *delayCondition* a *string* zeta, indicando que não há alterações no sistema que dependam do tempo (linha 25 – 26). Caso não seja nulo, será adicionado ao processo de escolhas externas uma última escolha formado pela variável *combinedNegation*, indicando a situação em que a passagem de tempo não altera o comportamento (linha 28).

Por fim, é gerado um evento composto informando o valor de cada variável eta. Este evento composto pode ser rotulado por *delay* ou *maxDelay*. Para saber qual usar, o processo DELAY recebe como argumento *isMaxDelay* e, dependendo se este

é verdadeiro ou falso, se comporta como o processo auxiliar $AUX\_MAXDELAY$ ou AUX_DELAY, respectivamente. A principal diferença entre esses dois processos está em que o processo AUX_DELAY modela o tempo corrente do sistema, enquanto o processo $AUX\_MAXDELAY$ trata de modelar o tempo total do sistema.

Algoritmo 4: Delay

Input: dfrs, etaBingind
Output: delayCSP

```
1  delayCSP = newString();
2  for function ∈ dfrs.F do
3      for discretGuard, timedGuard, actionList ∈ function do
4          if timedGuard ≠ null then
5              etaName = etaBinding.find((discretGuard, timedGuard));
6              etaSetFalse = etaSetFalse + etaName + "_false{" + etaName + " = false} -> ";
7  if etaSetFalse = null then
8      delayCSP = delayCSP + "DELAY(isMaxDelay) = Skip;(";
9  else
10     delayCSP = delayCSP + "DELAY(isMaxDelay) = " + etaSetFalse + "Skip;(";
11 for function ∈ dfrs.F do
12     for (discretGuard, timedGuard, actionList) ∈ function do
13         if timedGuard ≠ null then
14             if firstCondition then
15                 firstCondition = false;
16             else
17                 delayCondition = delayCondition + "[*]";
18             guardCode = changeFormat(discretGuard);
19             if combinedNegation = null then
20                 combinedNegation = "!(" + guardCode + ")";
21             else
22                 combinedNegation = combinedNegation + "&&!(" + guardCode + ")";
23             etaName = etaBinding.find((discretGuard, timedGuard));
24             delayCondition = delayCondition + "([" + guardCode + "]" + etaName + "_true{" +
               etaName + " = true} -> Skip)";
25 if delayCondition = null then
26     delayCondition = delayCondition + "zeta -> Skip);";
27 else
28     delayCondition = delayCondition + "[*]([" + combinedNegation + "]zeta -> Skip));";
29 delayCSP =
   delayCSP + delayCondition + "if(isMaxDelay){AUX_MAXDELAY()}else{AUX_DELAY()};";
30 delayCSP =
   delayCSP + generateAuxDelay(dfrs, etaBinding) + generateAuxMaxDelay(dfrs, etaBinding);
```

Para CSP_M, como é possível ver no exemplo 3.7, ao final há o canal *delayChannel*, onde é informado cada valor das variáveis *eta*. Porém, por uma própria limitação do CSP#, não é possível usar o parâmetro como um evento. Para contornar esta limitação, foi utilizado o parâmetro *isMaxDelay*, que é um parâmetro booleano e será usado para determinar a chamada de um dos eventos auxiliares AUX_DELAY e $AUX\_MAXDELAY$, mostrado na última linha do exemplo 3.8.

Os processos AUX_DELAY, mostrado no Algoritmo 5, e $AUX\_MAXDELAY$, mostrado no Algoritmo 6 possuem um comportamento semelhante, diferindo unicamente pelo nome do evento utilizado: *delay* e *maxDelay*, respectivamente.

Primeiramente, é declarada a variável de saída, inicializada com o nome do processo,

como pode ser visto nas linhas 1 e 2. Em seguida, são percorridas todas as funções do DFRS, verificando a relação ($discreteGuard \times timedGuard \rightarrow actionList$), onde, para cada guarda temporal não nula, identifica-se sua variável *eta* correspondente e adiciona a mesma na lista *etaListName* (linhas 3 – 7).

Se a lista *etaListName* for vazia, o evento composto será apenas *delay* para o processo *AUX_DELAY* e *maxDelay* para o processo *AUX_MAXDELAY*. Contudo, caso não seja vazia, percorre-se a lista *etaListName* para adicionar ao evento composto cada variável *eta*, seguido, por fim, pelo processo *Skip*. A representação desses dois processos auxiliares , no contexto da VM, podem ser vistos no exemplo 3.9.

Exemplo 3.9 – Representação de *AUX_DELAY* e *AUX_MAXDELAY* de VM

```
1 AUX_DELAY() = delay.eta1.eta2.eta3.eta4 -> Skip;
2
3 AUX_MAXDELAY() = maxDelay.eta1.eta2.eta3.eta4 -> Skip;
```

Algoritmo 5: Delay auxiliar

Input: dfrs, etaBingind
Output: auxDelayCSP

```
1  auxDelayCSP = newString();
2  auxDelayCSP = "AUX_DELAY() = ";
3  for function ∈ dfrs.F do
4      for discretGuard, timedGuard, actionList ∈ function do
5          if timedGuard ≠ null then
6              etaName = etaBinding.find((discretGuard, timedGuard));
7              etaListName.add(etaName);
8  if etaListName = null then
9      auxDelayCSP = auxDelayCSP + "delay -> Skip;";
10 else
11     auxDelay = auxDelay + "delay";
12     for eta ∈ etaListName do
13         auxDelayCSP = auxDelayCSP + "." + etaListName.get(i);
14     auxDelayCSP = auxDelayCSP + " -> Skip;";
```

3.2.5 Elementos auxiliares

Após representar a passagem de tempo, é necessário representar as transições funcionais, conforme descrito nos Algoritmos 7 e 8.

No Algoritmo 7, após a inicialização da variável de retorno, na linha 1, é feito apenas a indicação de que as transições funcionais se comportam como *SYSTEM_BEHAVIOR*, explicado anteriormente, que se trata do comportamento do sistema (linha 2).

No Algoritmo 8, na linha 1, variável de retorno é inicializada. Já na linha 2, inicializa-se a variável *funTrans* com o valor *false*. Em seguida, o processo *FUN* se comporta como o processo *FUN_TRANS*. Após o processo *FUN_TRANS* terminar com sucesso, verifica-se o valor da variável auxiliar *funTrans*. Se for verdadeira, indica que houve uma mudança

Algoritmo 6: MaxDelay auxiliar

Input: dfrs, etaBingind
Output: auxMaxDelayCSP

```
1  auxMaxDelayCSP = newString();
2  auxMaxDelayCSP = "AUX_MAXDELAY() = ";
3  for function ∈ dfrs.F do
4      for discretGuard, timedGuard, actionList ∈ function do
5          if timedGuard ≠ null then
6              etaName = etaBinding.find((discretGuard, timedGuard));
7              etaListName.add(etaName);
8  if etaListName = null then
9      auxMaxDelayCSP = auxMaxDelayCSP + "delay -> Skip; ";
10 else
11     auxDelay = auxDelay + "delay";
12     for eta ∈ etaSetFalse do
13         auxMaxDelayCSP = auxMaxDelayCSP + "." + etaListName.get(i);
14     auxMaxDelayCSP = auxMaxDelayCSP + " -> Skip; ";
```

Algoritmo 7: Function Transition

Input:
Output: funTransCSP

```
1  funTransCSP = newString();
2  funTransCSP = "FUN_TRANS() = SYSTEM_BEHAVIOR(); ";
```

no estado do sistema, caracterizada pela ocorrência de uma transição funcional. Esse comportamento se repete até que *funTrans* seja *false*. Neste caso, diz-se que o sistema atingiu um estado estável e agora o tempo irá passar, seguido do recebimento de novos valores de entrada. Antes de permitir o tempo passar, o processo FUN irá se comportar como $OUTPUTS$. Este processo é descrito a seguir.

Algoritmo 8: Representação de Function

Input:
Output: funCSP

```
1  funCSP = newString();
2  funCSP = "FUN() = funTrans_false{funTrans = false} ->
   FUN_TRANS(); if a(funTrans){FUN()}else{OUTPUTS()}; ";
```

O Algoritmo 11 é responsável por gerar o código CSP# associado ao processo $OUTPUTS$. Este algoritmo cria um evento composto chamado *output*, visto na linha 2, após inicializar a variável de retorno. Para criar este evento composto, é necessário varrer todas as variáveis de saída do DFRS para adicionar ao evento (linhas 3 – 4). Após essa varredura, adiciona-se o processo *Skip* à *string* de retorno (linha 5).

É possível ver no exemplo 3.11 o código CSP# gerado para o exemplo da VM, enquanto no exemplo 3.10 se tem a representação em CSP_M. Estes códigos se diferenciam basicamente pela sintaxe de CSP# e CSP_M. O processo que mais se diferencia, além da sintaxe, é o processo *FUN_TRANS*. Como mencionado anteriormente, no processo de *SYSTEM_BEHAVIOR* em CSP_M é necessário passar todas as variáveis do sistema

Algoritmo 9: Outputs

Input: dfrs
Output: outputCSP

```
1 outputCSP = newString()
2 outputCSP = outputCSP + "OUTPUTS() = output";
3 for variable ∈ dfrs.O do
4     outputCSP = outputCSP + "." + variable.name;
5 outputCSP = outputCSP + " -> Skip;";
```

utilizadas como parâmetro. No processo *FUN_TRANS*, como ele se comporta como *SYSTEM_BEHAVIOR*, é necessário, em primeiro lugar, carregar todas as variáveis através dos canais de *get*, antes de passá-las como argumentos para o processo. Processos como *FUN* e *OUTPUTS*, nesse aspecto de uso de variáveis, apenas carregam as variáveis utilizadas para usá-las em seus processos, como o valor de *funTrans* e os valores das variáveis de saída para a composição do evento composto.

Exemplo 3.10 – Representação em CSP_M de *Function Transition* e de *Outputs* de VM

```
1  FUN_TRANS =
2  get!the_system_mode?I_the_system_mode.v_the_system_mode ->
3  get!eta1?B.v_eta1 ->
4  (...) ->
5  SYSTEM_BEHAVIOUR(...)
6
7  FUN = set!funTrans!B.false ->
8  FUN_TRANS
9  ; get!funTrans?B.engaged ->
10 if engaged then FUN else OUTPUTS
11
12 OUTPUTS =
13 get!the_system_mode?v_the_system_mode ->
14 get!the_coffee_machine_output?v_the_coffee_machine_output ->
15 output.the_system_mode.v_the_system_mode.(...) -> SKIP
```

Exemplo 3.11 – Representação em CSP# de *Function Transition* e de *Outputs* de VM

```
1  FUN_TRANS() = SYSTEM_BEHAVIOUR();
2
3  FUN() = funTrans_false{funTrans = false} -> FUN_TRANS();
4  ifa(funTrans){
5  FUN()
6  } else {
7  OUTPUTS()
8  };
9
10 OUTPUTS() = output.the_system_mode.the_coffee_machine_output -> Skip;
```

Para finalizar a geração do código CSP# que representa a alternância de transições funcionais e de atraso de um DFRS, cria-se o processo $SPECIFICATION$, que representa o comportamento cíclico de um DFRS, como pode ser visto no Algoritmo 10.

Na linha 1, a variável de retorno é inicializada, enquanto que na linha seguinte atribui-se o nome do processo em questão à variável de retorno. Também é necessário inicializar o temporizadores, caso eles existam no sistema, atribuindo a eles o valor *false* (linhas 3 – 5). Em seguida, declara-se o corpo do processo $SPECIFICATION$, criando

um processo recursivo que realiza alternadamente transições funcionais e de atraso, como é mostrado na linha 6.

Algoritmo 10: Representação de Especificação

Input: dfrs
Output: specCSP

```
1 specCSP = newString();
2 specCSP = "SPECIFICATION() = ";
3 if dfrs.T ≠ null then
4     for variable ∈ dfrs.T do
5         specCSP = specCSP + variable.name + "_false{" + variable.name + " = false} -> ";
6 specCSP = specCSP + "FUN(); stableState -> delayTransition ->
  DELAY(true); INPUTS; DELAY(false); SPECIFICATION(); ";
```

Por fim, como último passo, cria-se o processo S, cujo comportamento é igual ao do processo $SPECIFICATION$, no entanto, escondendo eventos internos que não caracterizam entradas e saídas do sistema (linhas 3 – 5). No exemplo 3.13 é possível ver o código gerado em CSP# no contexto da VM. Para CSP_M, explicitado no exemplo 3.12, a principal diferença é a definição de um processo chamado *SYSTEM*, onde se trata de uma composição paralela entre os processos de *SPECIFICATION* e *SYSTEM_MEMORY*, onde é informado que esses eventos são síncronos nos canais de *get* e *set*, utilizados para as operações de leitura e atualização das variáveis do sistema.

Algoritmo 11: Representação do Sistema

Input: reqList
Output: specSystemCSP

```
1 specSystemCSP = newString();
2 specSystemCSP = "SYSTEM() = SPECIFICATION(){";
3 for requirement ∈ reqList do
4     specSystemCSP = specSystemCSP + requirement + ",";
5 specSystemCSP = specSystemCSP + "stableState, delayTransition}; ";
```

Exemplo 3.12 – Representação em CSP_M de Especificação e do Sistema

```
1 SPECIFICATION =
2 set!the_request_timer!B.false ->
3 FUN ; stableState ->
4 delayTransition -> DELAY(maxDelay) ; INPUTS ; DELAY(delay)
5 ; SPECIFICATION
6
7 SYSTEM = SPECIFICATION [| {|get,set|} |] SYSTEM_MEMORY
8
9 S = SYSTEM \ {| ... |}
```

Exemplo 3.13 – Representação em CSP# de Especificação e do Sistema

```
1 SPECIFICATION() = the_request_timer_false{the_request_timer = false} -> FUN();
2 stableState -> delayTransition -> DELAY(true); INPUTS; DELAY(false);
3 SPECIFICATION();
4
5 S() = SPECIFICATION()
6 \ {REQ003, REQ005, REQ002, REQ001, REQ004, stableState, delayTransition};
```

Desta forma, conclui-se a explicação do passo-a-passo necessário para gerar especificações CSP# que simulam o comportamento de modelos formais DFRS. No próximo capítulo será realizado uma análise empírica a partir das representações do DFRS em CSP_M, fornecida pelo NAT2TEST, e em CSP#, fornecida a partir do passo-a-passo explicado anteriormente nas ferramentas FDR e PAT para a verificação de propriedades de processos CSP.

4 Análise Empírica

Este capítulo tem como finalidade descrever a análise empírica com representações do DFRS em CSP_M, fornecida pelo NAT2TEST, e CSP#, gerada a partir do passo-a-passo explicado no capítulo anterior, de tal forma que possa ser verificado aspectos de desempenho das representações sendo executadas nas ferramentas FDR e PAT para analisar propriedades clássicas de processos CSP. Porém, é preciso, antes de apresentar os resultados desta análise, listar os exemplos considerados, além de comentar as limitações desta análise empírica.

4.1 Exemplos utilizados

Este trabalho considerou três sistemas diferentes, de forma a observar particularidades que cada sistema possui no momento de representar em CSP.

- *Vending Machine*: já explicada no Capítulo 2, simula uma máquina de café, onde o usuário coloca uma moeda e aperta o botão para retirar café, onde, de acordo com a demora em pressionar o botão, o café produzido pode ser fraco ou forte.
- *Nuclear Power Plant* (NPP): esse sistema trata de uma versão simplificada de um sistema de controle de segurança de uma central nuclear (LEONARD; HEITMEYER, 2003). O sistema monitora a pressão da água, e se a pressão da água for considerada baixa, o sistema refrigera o reator nuclear.
- *Turn Indicator System* (TIS): esse sistema foi desenvolvido a partir de um sistema automotivo que lida com o indicador de luzes dos carros da Mercedes (PELESKA et al., 2011). O modelo trata de uma versão simplificada do sistema real, servindo para prova de conceito, já que esta versão representa parte de um sistema crítico real com aspectos concorrentes.

4.2 Limitações

A execução das ferramentas FDR e PAT foram feitos na mesma máquina, utilizando as seguintes configurações:

1. CSP# utilizando PAT:
    a) Processador Core i5

b) Memória RAM 6GB

c) Sistema operacional Windows 8 x64

2. CSP$_M$ utilizando FDR:

a) Processador Core i5

b) Memória RAM 4GB

c) Sistema operacional Linux - Ubuntu 14.04 x64

Apesar dos testes utilizando o FDR terem sido feitos na mesma máquina utilizada para os testes em PAT, vale salientar que eles foram feitos utilizando uma máquina virtual instalada. Portanto, é preciso ter em mente que é possível haver um maior consumo de tempo para a verificação de propriedades, visto que o desempenho cai pelo fato de haver menos memória RAM alocada para o FDR e o sistema operacional ter sido simulado.

Por outro lado, sabe-se que a verificação de propriedades das representações em CSP# é feita na ferramenta PAT. Mesmo as execuções não serem feitas em ambientes simulados, é preciso ter em mente que a ferramenta possui interface gráfica, o que pode acabar aumentando o tempo de verificação das propriedades, uma vez que é necessário tempo para exibir na tela os resultados para o usuário.

4.3 Resultado da análise

Para cada um dos exemplos citados anteriormente (VM, NPP e TIS) descritos foram feitas diversas execuções a fim de coletar dados sobre desempenho. Além de observar o desempenho, também foi feita a análise da quantidade de estados percorridos durante a verificação de propriedades clássicas do CSP.

4.3.1 Análise de estados

Na análise de estados, cada exemplo foi rodado uma vez para averiguar a quantidade de estados e de transições, como pode ser visto na Tabela 2.

Tabela 2 – Quantidade de estados para FDR e PAT
[Fonte: Elaboração própria]

FDR			
	VM	NPP	TIS
Número de estados	4648	15367	165354
Número de transições	4756	16302	174542
PAT			
	VM	NPP	TIS
Número de estados	1716	8210	68053
Número de transições	1786	8947	73998

Pelos resultados mostrado na Tabela 2, a análise indica que CSP# tem uma representação mais concisa, por considerar menos estados e menos transições. No entanto, é importante salientar que não se tem uma prova formal que o código CSP# gerado possui a mesma semântica do código CSP_M correspondente. Este aspecto é considerado como um dos trabalhos futuros.

4.3.2 Análise de propriedades

Nesta análise, propriedades clássicas de processos CSP foram avaliadas no contexto dos códigos gerados em CSP_M e CSP#. Para os três sistemas antes mencionados, foram analisadas as seguintes propriedades: se o sistema é live de *deadlock*, se o sistema é determinístico e se o sistema é livre de divergências. Para a verificação dessas propriedades, foram usadas os seguintes *asserts*:

1. Para CSP_M:
   a) *assert S :[deadlock free [F]]*
   b) *assert SYSTEM :[deterministic [FD]]*
   c) *assert S :[livelock free [FD]]*
2. Para CSP#:
   a) *#assert S deadlockfree;*
   b) *#assert S deterministic;*
   c) *#assert S divergencefree;*

Em CSP_M, a análise de determinismo é realizada em *SYSTEM* e não em *S*, pois *S*, ao esconder eventos internos introduz não-determinismo. A Tabela 3 apresenta o resultado desta análise.

Tabela 3 – Análise de propriedades para FDR e PAT
[Fonte: Elaboração própria]

FDR			
	VM	NPP	TIS
Ausência de *deadlock*	*true*	*true*	*true*
Não-determinismo	*true*	*false*	*false*
Ausência de divergência	*true*	*false*	*true*
PAT			
	VM	NPP	TIS
Ausência de *deadlock*	*true*	*true*	*true*
Não-determinismo	*true*	*false*	*false*
Ausência de divergência	*true*	*true*	*true*

É possível notar uma diferença entre os resultados feitos para os dialetos CSP_M e CSP#. Um caso claro deste ocorrido é a verificação da propriedade de ausência de

divergência, onde para CSP_M é *false*, mas para mas para CSP# é *true*: enquanto o código CSP# do NPP é livre de divergência, o código CSP_M não é. Esta diferença é provavelmente uma consequência de uma limitação deste trabalho, pois na definição do processo S, alguns eventos que deveriam ser escondidos de *SYSTEM* ainda são visíveis. Eventos auxiliares como a rastreabilidade das funções com os requisitos (como REQ001 até REQ005) e outros que compõe o comportamento cíclico do DFRS (como delayTransition e stableState) são escondidos para que não sejam exibidos nas funcionalidades de verificação e simulação da ferramenta PAT a partir do CSP#. Contudo, eventos auxiliares utilizados durante processos como o *SYSTEM_BEHAVIOR* (eventos utilizados para atribuições de variáveis) não foram escondidos. Este é um trabalho futuro. Portanto, é possível que após esta correção o processo S em CSP# se torne divergente pela presença de um laço de eventos internos.

4.3.3 Análise de desempenho

Para analisar o desempenho em CSP, foram realizadas 30 execuções para cada exemplo e análise de propriedade. A ideia é coletar o resultado de cada uma das trinta verificações para cada exemplo e fazer uma média de tempo entre todas as execuções. Para isso foi calculada a média aritmética somando todos os valores e dividindo pela quantidade de testes rodados, como pode ser visto na fórmula a seguir:

$$M = \frac{1}{N}\sum_{i=1}^{N} t_i$$

Após a obtenção das médias, elas foram utilizadas para o cálculo do desvio padrão. Esse desvio serve para indicar a variação com relação a média de tempo de análise para cada propriedade. Esse desvio pode ser calculado a partir da seguinte fórmula:

$$DP = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(t_i - \bar{t})^2}$$

Os resultados da análise de desempenho são mostrados nas Tabelas 4, 5 e 6. É possível destacar que essas verificações são dependentes do resultado da análise de propriedades, pois é necessário explorar todos os estados em CSP_M, diferentemente de CSP# (como ocorreu com NPP para o caso de divergência). O momento agora é observar o tempo gasto pelas ferramentas PAT e FDR para verificar cada propriedade. Vale salientar que em cada tabela de resultado, os tempos descritos estão sendo retratados em segundos.

Apesar das limitações citadas na seção anterior, os resultados obtidos indicam que FDR precisou de mais tempo do que PAT para realizar as análises consideradas. Provavelmente, uma consequência da representação com mais estados em CSP_M. Ou seja,

Tabela 4 – Desempenho da análise de ausência de *deadlock* com FDR e PAT
[Fonte: Elaboração própria]

FDR			
	VM	NPP	TIS
Média de tempo entre as execuções	1,869666667	0,906333333	25,23666667
Desvio padrão	0,018659071	0,017515182	0,0149328
PAT			
	VM	NPP	TIS
Média de tempo entre as execuções	0,023628237	0,053011127	0,47671529
Desvio padrão	0,007392767	0,003060243	0,043173855

Tabela 5 – Desempenho da análise de determinismo com FDR e PAT
[Fonte: Elaboração própria]

FDR			
	VM	NPP	TIS
Média de tempo entre as execuções	2,08	1,075	28,67266667
Desvio padrão	0,020171677	0,013582443	0,016174338
PAT			
	VM	NPP	TIS
Média de tempo entre as execuções	0,022353157	0,003199763	0,001320503
Desvio padrão	0,005192472	0,000720841	0,000319756

Tabela 6 – Desempenho da análise de ausência de divergência com FDR e PAT
[Fonte: Elaboração própria]

FDR			
	VM	NPP	TIS
Média de tempo entre as execuções	1,929	0,839	25,553
Desvio padrão	0,015165751	0,015391444	0,024516005
PAT			
	VM	NPP	TIS
Média de tempo entre as execuções	0,033496813	0,092593157	0,891976367
Desvio padrão	0,013142618	0,003380782	0,019798772

para verificar uma mesma propriedade, em CSP#, percorre-se uma menor quantidade de estados.

Portanto, com estes resultados, conclui-se que este trabalho alcançou o seu objetivo de gerar automaticamente uma representação de DFRSs (em CSP#) mais concisa e menos suscetível ao problema de explosão de estados.

5 Considerações Finais

Realizar teste de sistemas complexos pode ser difícil devido a sua complexidade e criticidade. Porém, ferramentas para dar suporte a geração de casos de testes têm sido desenvolvidas, e entre elas está a estratégia NAT2TEST. Essa estratégia utiliza linguagem natural controlada para gerar casos de teste automaticamente.

A estratégia também consegue representar formalmente sistemas reativos baseados em fluxo de dados utilizando a álgebra de processo CSP, no dialeto CSP_M. Porém, devido ao problema de explosão de estados que é causado por esse dialeto, este trabalho propôs a representação de sistemas utilizando CSP#, outro dialeto de CSP.

Ao longo deste trabalho, foi apresentada uma forma de gerar especificações CSP# para esses sistemas. A implementação criada neste trabalho deve ser posteriormente integrada à ferramenta NAT2TEST, o que facilitará a análise de sistemas mais complexos do que aqueles utilizados atualmente, além da geração de mais casos de teste a partir dos modelos CSP# gerados.

Além de mostrar a forma como se cria a representação em CSP#, foi feita uma análise empírica em cima de alguns exemplos para observar as vantagens e desvantagens de se gerar a especificação do DFRS em CSP#, apontando também diferenças em relação a representação em CSP_M.

Apesar de não poder afirmar que um dialeto é melhor do que o outro em situações gerais, além de precisar considerar as limitações associadas às análises realizadas neste trabalho, os resultados quantitativos obtidos indicam que a representação em CSP# é potencialmente mais concisa. Permitindo, assim, a análise de propriedades e geração de testes no contexto de sistemas mais complexos do que os hoje considerados.

O uso destas técnicas antes do desenvolvimento de *software* complexos pode trazer um maior entendimento do que será feito, a fim de evitar eventuais problemas que possam ocorrer no sistema. Ao final do desenvolvimento, é esperado um *software* com maior qualidade e com menor probabilidade de falhas.

5.1 Trabalhos futuros

Como continuação deste trabalho, destacam-se os seguintes trabalhos futuros:

- Correções de erros menores no algoritmo de geração do CSP#, como alguns eventos que não foram escondidos na definição de S e adequar o nome das variáveis com "-" para "_", já que o dialeto entende "-" como um operador aritmético;

- Refazer a análise empírica executando FDR em um ambiente não virtualizado, ou considerar a recente versão para Windows do FDR;
- Realizar experimentos com outros exemplos; preferencialmente, exemplos de maior complexidade;
- Integrar o código que gera a representação em CSP# à estratégia NAT2TEST;
- Implementar uma estratégia de geração de testes a partir da representação em CSP#;
- Avaliar através da geração de testes se os testes que podem ser gerados a partir do código CSP_M também podem ser gerados a partir do código CSP#. Apesar de não ser uma prova formal, esta ação permitirá ter uma maior noção se a semântica associada ao código CSP# gerado é semelhante a do código CSP_M.

Referências

ALMEIDA, J. B. et al. *Rigorous software development: an introduction to program verification*. [S.l.]: Springer Science & Business Media, 2011.

BODDU, R. et al. RETNA: from requirements to testing in a natural way. In: IEEE. *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*. [S.l.], 2004. p. 262–271.

CARVALHO, G. et al. A Formal Model for Natural-Language Timed Requirements of Reactive Systems. In: *Formal Methods and Software Engineering*. [S.l.]: Springer International Publishing, 2014. p. 43–58.

CARVALHO, G. et al. NAT2TESTSCR: Test case generation from natural language requirements based on SCR specifications. *Science of Computer Programming*, Elsevier, v. 95, p. 275–297, 2014.

CARVALHO, G.; SAMPAIO, A.; MOTA, A. A CSP Timed Input-Output Relation and a Strategy for Mechanised Conformance Verification. In: *Formal Methods and Software Engineering*. [S.l.]: Springer Berlin Heidelberg, 2013. p. 148–164.

FILLMORE, C. J. The case for case. 1967.

FORMALSYSTEMS. *Formal Systems Website*. 1986–2010. Disponível em: <http://www.fsel.com>.

GOMES, L. F. d. S. Redes de Petri reactivas e hierárquicas-integração de formalismos no projecto de sistemas reactivos de tempo-real. FCT-UNL, 1997.

LARSEN, K. G.; MIKUCIONIS, M.; NIELSEN, B. Online testing of real-time systems using uppaal. In: *Formal Approaches to Software Testing*. [S.l.]: Springer Berlin Heidelberg, 2005. p. 79–94.

LEONARD, E. I.; HEITMEYER, C. L. Program Synthesis from Formal Requirements Specifications Using APTS. *Higher Order Symbol. Comput.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 16, n. 1-2, p. 63–92, mar. 2003. ISSN 1388-3690. Disponível em: <http://dx.doi.org/10.1023/A:1023072104553>.

NOGUEIRA, S.; SAMPAIO, A.; MOTA, A. Test generation from state based use case models. *Formal Aspects of Computing*, Springer London, v. 26, n. 3, p. 441–490, 2014.

PELESKA, J. et al. A Real-World Benchmark Model for Testing Concurrent Real-Time Systems in the Automotive Domain. In: WOLFF, B.; ZAÃ¯DI, F. (Ed.). *Testing Software and Systems*. Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, v. 7019). p. 146–161. ISBN 978-3-642-24579-4. Disponível em: <http://dx.doi.org/10.1007/978-3-642-24580-0_11>.

ROSCOE, A. *The Theory and Practice of Concurrency*. [S.l.]: Prentice-Hall, 1998.

SCHWITTER, R. English as a formal specification language. In: IEEE. *Database and Expert Systems Applications, 2002. Proceedings. 13th International Workshop on*. [S.l.], 2002. p. 228–232.

SHI, L. et al. An analytical and experimental comparison of CSP extensions and tools. In: *Formal Methods and Software Engineering*. [S.l.]: Springer Berlin Heidelberg, 2012. p. 381–397.

SUN, J.; LIU, Y.; DONG, J. S. Model checking CSP revisited: Introducing a process analysis toolkit. *Leveraging Applications of Formal Methods, Verification and Validation*, Springer, p. 307–322, 2009.

WOODCOCK, J. et al. Formal methods: Practice and experience. *ACM Computing Surveys (CSUR)*, ACM, v. 41, n. 4, p. 19, 2009.