UNIVERSIDADE FEDERAL DE PERNAMBUCO

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
CENTRO DE INFORMÁTICA

2013.1

VERIFICAÇÃO DE CONFORMIDADE A PARTIR DE MODELOS EM CSP DERIVADOS DE REQUISITOS EM CNL

TRABALHO DE GRADUAÇÃO

Aluno: Eduardo Bastos Rocha (ebr@cin.ufpe.br)
Orientador: Augusto Cezar Alves Sampaio (acas@cin.ufpe.br)

Recife, Setembro de 2013

Agradecimentos

Este trabalho foi construído com a ajuda de muitas pessoas, e aqui eu gostaria de registrar meus agradecimentos:

- Agradeço à minha família, por todo o apoio e compreensão. Sem eles, eu não chegaria aqui;
- Agradeço aos meus amigos pelo companheirismo e por me alegrar nas horas que eu mais precisava;
- Agradeço ao aluno de doutorado Gustavo Henrique Porto de Carvalho por toda a orientação e conselhos sobre como implementar a ferramenta;
- Agradeço ao também aluno de doutorado Sidney Nogueira e a Hugo Leonardo pela ajuda para usar a ferramenta ATG.

Agradeço a todos os professores e funcionários do Centro de Informática, em especial ao professor Augusto Sampaio por me orientar no andamento deste trabalho.

Resumo

Sistemas de software estão cada vez mais crescendo em tamanho e complexidade, e frequentemente eles não se comportam da maneira esperada. Quando falhas surgem, prejudicam o usuário direta ou indiretamente. Todos estes fatores demandam melhores processos para verificar e garantir a corretude de um sistema. Relação de conformidade é uma técnica que representa a noção de conformidade entre uma implementação e uma especificação. Ela é usada para verificar se uma implementação está conforme com sua especificação, e assim verificar corretude.

O objetivo deste trabalho é de implementar uma relação de conformidade entre um modelo da implementação e um modelo da especificação de um sistema. Os modelos serão implementados em CSP e serão derivados a partir de requisitos descritos em uma *Controlled Natural Language* (CNL).

Sumário

1. Introdução

Sistemas de software estão crescendo cada vez mais em tamanho e complexidade e frequentemente eles não se comportam da maneira esperada. Falhas surgem com frequência e causam perdas direta ou indiretamente aos usuários. Todos estes fatores demandam melhores processos para verificar a corretude do sistema.

Para atingir um alto grau de confiabilidade na produção de um sistema são usadas geralmente as técnicas de Verificação Formal e *Model Based Testing* (MBT). A Verificação Formal utiliza uma abordagem baseada em lógica matemática para demonstrar formalmente que um programa implementa corretamente as funcionalidades especificadas. *Model Based Testing* visa a criação de testes, para verificação do sistema, a partir de modelos que representam o comportamento esperado do sistema.

Abordagens com testes formais também foram propostas para comparar, através de uma relação de conformidade, o comportamento de uma implementação com respeito à sua especificação. Informalmente, se algum teste deve ser aceito por uma especificação, esse teste também deve ser aceito pela implementação. Além disso, qualquer teste que possa não ser aceito pela implementação, também deve poder não ser aceito na especificação [10].

Em uma relação de conformidade, a especificação do sistema deve ser provida através de um modelo formal, como Máquinas de Estado Finitas (*Finite State Machines*, FSM). A implementação deve também ser modelada usando a mesma representação formal. A conformidade é definida, portanto, como uma relação matemática entre os modelos.

Em [3] foi proposta uma relação de conformidade chamada CSPTIO, baseada na álgebra de processos CSP. Essa relação é capaz de analisar sistemas que interagem com seu ambiente através de conjuntos de eventos de entradas e saídas. Os eventos de entrada e saída obedecem a restrições de tempo. Esse tipo de sistema é também chamado de *Data Flow Reactive System* (DFRS).

Em CSPTIO, a especificação e a implementação são descritos em CSP. A verificação de conformidade é feita com o auxílio das ferramentas FDR [6], para checar refinamentos, e Z3 [7], uma ferramenta para resolver problemas SMT.

1.1. Objetivos

Em [1] foi proposta uma ferramenta (NAT2TEST) para a criação de vetores de testes para sistemas a partir dos requisitos do sistema descritos em linguagem natural. Os requisitos são especificados através de uma Linguagem Natural Controlada (*Controlled Natural Language*, CNL). Inicialmente, é feita

uma análise sintática dos requisitos para verificar a conformidade com a estrutura da linguagem CNL. Em seguida, é gerada uma interpretação semântica dos requisitos seguindo a estrutura de *Verb Case Frames* (CF).

O objetivo deste trabalho de graduação é de estender esta ferramenta com a funcionalidade de verificação de conformidade CSPTIO de forma automatizada. As principais contribuições deste trabalho serão:

1. Geração de código CSP, a partir dos Case Frames, para a especificação do sistema e a implementação;
2. Integração com FDR para realizar a verificação de conformidade CSPTIO.

2. Conceitos Básicos

Nesta seção serão introduzidos alguns conceitos básicos para o entendimento deste trabalho.

2.1. CSP

CSP (Communicating Sequential Processes) é uma linguagem formal concebida para descrever aspectos comportamentais de sistemas. O componente básico de CSP é o processo. CSP possui dois processos primitivos: um que representa um fim com sucesso (SKIP) e um que representa um fim anormal (STOP), que também pode ser interpretado como um *deadlock*. Um processo é definido em termos de eventos. Eventos são atômicos e representam a interface pela qual o sistema se comunica com o ambiente ou com outros processos. Quando um evento ocorre, uma transição é disparada, e então, o processo muda para um novo estado. Para definir um processo como uma sequência de eventos, usamos a notação de evento prefixo ($ev \rightarrow$ P), onde *ev* é um evento e P, um processo. É possível também criar processos recursivos, como por exemplo $P = a \rightarrow b \rightarrow P$.

CSP também permite a definição de escolhas. Para isso existem os operadores de escolha externa, interna e condicional. A escolha externa (□) representa uma escolha determinística entre dois processos. A escolha interna (⊓) representa uma escolha não-determinística. A escolha condicional (*if*) é similar às estruturas condicionais de linguagens de programação padrão.

Dois outros operadores relevantes são o de composição sequencial e de composição paralela de processos. A seguinte composição sequencial *P = P1; P2*, por exemplo, mostra que o comportamento de P é equivalente ao comportamento de P1 seguido do comportamento de P2, quando P1 é finalizado com sucesso. A composição paralela pode ser com (| |) ou sem (| | |) sincronização entre os processos compostos. Processos CSP sincronizam-se entre si através de eventos. O termo *c!e*, onde *c* é um canal, denota um evento resultante da avaliação de *e*, que pode ser qualquer expressão CSP válida. O termo *c?v* representa qualquer valor de um conjunto particular (por exemplo, para *channel c : {0,1,2}*, *c?v* significa *c.0*, *c.1*, *c.2*). Também é possível interpretar os símbolos *!* e *?* como envio ou recebimento de valores pelo canal, respectivamente.

2.2. NAT2TEST

NAT2TEST é uma ferramenta que foi desenvolvida na tese de Doutorado do aluno Gustavo Henrique Porto de Carvalho. Seu objetivo é de trazer o uso de MBT para as fases iniciais do projeto. A estratégia NAT2TEST é de gerar casos de teste a partir de requisitos escritos em uma *Controlled Natural Language* (CNL). A estratégia é direcionada a sistemas reativos baseados em fluxos de dados (*Data-Flow Reactive Systems*, DFRS).

A estratégia NAT2TEST é composta por quatro fases. Primeiramente, os requisitos são analisados sintaticamente de acordo com SysReq-CNL, uma CNL proposta para descrever sistemas reativos baseados em fluxos de dados. Então, os requisitos são analisados semanticamente utilizando a teoria de Gramática de Casos, nesse ponto são gerados os *Case Frames*. Em seguida, uma representação formal em CSP é derivada a partir dos *Case Frames*. Finalmente, a relação de conformidade CSPTIO é usada para gerar os casos de teste. Cada fase do processo é feita por um módulo do sistema. Os módulos *CNL-Parser*, *CF-Generator* e *CSP-Generator* implementam as três primeiras fases. A geração de testes é feita com o auxílio das ferramentas FDR [6], um verificador de refinamentos, e Z3 [7], uma ferramenta para resolver problemas SMT.

3. Geração de CSP a partir de Case Frames

Esse processo é feito em duas etapas. Primeiro, geramos um *Data-Flow Reactive System* (DFRS) a partir dos *Case Frames* (CF's). Depois, geramos o código CSP a partir do DFRS. Cada etapa será detalhada nas próximas subseções.

Os algoritmos para geração de um DFRS a partir de Case Frames e para geração de CSP a partir de um DFRS foram idealizados pelo aluno de doutorado Gustavo Henrique Porto de Carvalho.

3.1. Geração de um DFRS a partir de Case Frames

3.1.1. A notação de Case Frames

Para gerar o DFRS que captura o comportamento esperado do sistema, precisamos inferir a semântica por trás dos requisitos descritos na CNL. Em [1], foi utilizada a teoria linguística de *Case Grammar* [2] para representar a semântica da CNL. Nessa teoria, as sentenças são interpretadas em termos dos papéis temáticos que cada palavra (ou grupo de palavras) exerce na sentença (e.g., Agente, Paciente, Local, Instrumento).

Na teoria de *Case Grammar*, cada verbo é associado com papéis temáticos específicos, formando assim os *Case Frames* (CF) baseados em verbos. Portanto, um CF é uma estrutura com espaços, que representam papéis temáticos, que serão preenchidos por elementos da sentença. Papéis podem ser opcionais ou obrigatórios em um CF.

No modelo definido em [1], 9 papéis temáticos foram considerados suficientes para capturar a semântica dos requisitos descritos na CNL. Como os requisitos têm a forma de ações que são executadas caso determinadas condições sejam verdadeiras, foram considerados papéis temáticos específicos para condições e ações.

Os papéis temáticos adotados para ações são:

- Action (ACT): a ação que será executada se as condições forem satisfeitas;
- Agent (AGT): entidade que realiza a ação;
- Patient (PAT): entidade que é afetada pela ação;
- To Value (TOV): o valor que o Paciente deve receber depois da ação.

Os papéis temáticos adotados para condições são:

- Condition Action (CAC): a ação que o Paciente da Condição deve executar para que a condição seja verdadeira;
- Condition Patient (CPT): o Paciente da Condição. É o elemento associado às condições;
- Condition From Value (CFV): o valor antigo do Paciente da Condição;
- Condition To Value (CTV): o valor do Paciente da Condição que satisfaz a condição;
- Condition Modifier (CMD): algum modificador relacionado à condição.

A tabela 1 mostra os Case Frames para o seguinte requisito:

REQ003 – *When the voltage is greater than 80, and the flashing timer is greater than or equal to 220, and the left indication lights are off, and the right indication lights are off, and the flashing mode is left flashing or the flashing mode is left tip flashing, the lights controller component shall: assign on to the left indication lights, assign off to the right indication lights, reset the flashing timer.* [Turn Indicator System from Mercedes]

Tabela 1 - Exemplo de Case Frames (Turn Indicator System)

Condition #1 – Main Verb (CAC): is			
CPT:	the voltage	CFV:	-
CMD:	greater than	CTV:	80
Condition #2 – Main Verb (CAC): is			
CPT:	the flashing timer	CFV:	-
CMD:	greater than or equal	CTV:	220
Condition #3 – Main Verb (CAC): are			
CPT:	the left indication lights	CFV:	-
CMD:	-	CTV:	off
Condition #4 – Main Verb (CAC): are			
CPT:	the right indication lights	CFV:	-
CMD:	-	CTV:	Off
Condition #5 – Main Verb (CAC): is			
CPT:	the flashing mode	CFV:	-
CMD:	-	CTV:	left flashing
OR – Main Verb (CAC): is			
CPT:	the flashing mode	CFV:	-
CMD:	-	CTV:	left tip flashing
Action – Main Verb (ACT): assign			
AGT:	the lights controller component	TOV:	on
PAT:	the left indication lights		
Action – Main Verb (ACT): assign			
AGT:	the lights controller component	TOV:	off
PAT:	the right indication lights		
Action – Main Verb (ACT): reset			
AGT:	the lights controller component	TOV:	-

PAT:	the flashing timer		

3.1.2. Definição de DFRS

DFRS's (*Data-Flow Reactive Systems*) são sistemas que interagem com o ambiente de uma forma contínua, por meio de um conjunto de eventos de entrada e de saída, enquanto obedecem a algumas restrições de tempo [5]. Um DFRS é definido formalmente da seguinte forma:

Um DFRS é uma tupla (I, O, T, B, b0, gc, F, M), onde:

- I é um conjunto não-vazio de variáveis tipadas (inteiros, booleanos, ponto flutuante) de entrada;
- O é um conjunto não-vazio de variáveis tipadas (inteiros, booleanos, ponto flutuante) de saída;
- T é um conjunto de timer's tipados (inteiros, ponto flutuante) ($I \cap O \cap T = \emptyset$);
- B é um conjunto de possíveis valores para as variáveis de entrada/saída e timers;
- b0 é um conjunto de valores iniciais para as variáveis de entrada/saída e timers;
- gc é o clock global do sistema;
- F é um conjunto de funções, uma para cada componente reativo do sistema. Cada função é definida por uma guarda (ou condição) e um conjunto de ações. A guarda é composta por dois conjuntos, um conjunto de expressões booleanas discretas e um conjunto de expressões booleanas referentes à tempo.
- M representa o modelo de estados e transições. M é construído a partir de F e de um estado inicial.

O modelo de DFRS usado neste trabalho é baseado em relações de transição similares às usadas no RT-Tester [4]. As relações de transição do RT-Tester relacionam estados pré e pós através de predicados sobre símbolos, que podem ser variáveis do sistema ou timers. Os símbolos podem ser *primed* ou *unprimed*. Símbolos *unprimed* estão relacionados com valores no estado pré, enquanto que símbolos *primed* referem-se a valores no estado pós. Essa relação evidencia dois tipos de transições: *discretas* e *temporais*. *Transições discretas* ocorrem sempre que o sistema tem uma transição habilitada, e sua execução gera efeitos colaterais, que alteram o estado do sistema (memória). *Transições temporais* ocorrem quando não há mais transições discretas habilitadas. A separação entre transições temporais e discretas permite que o tempo de execução do sistema

possa avançar sem que o estado do sistema seja alterado. Uma definição formal desta relação de transição pode ser encontrada em [4].

3.1.3. De CF para DFRS

O processo para criar um DFRS a partir de CF's consiste em três passos. Primeiro, as variáveis de entrada, saída e timers do sistema devem ser identificadas. Depois, as funções que definem o comportamento do sistema são definidas. E finalmente, o DFRS é criado.

Em um DFRS, as variáveis são de três possíveis tipos: entrada, saída e timer. Além disso, o nosso modelo suporta apenas três tipos de dados: inteiros, números de ponto flutuante e booleanos.

Entradas são as variáveis que são providas ao sistema pelo ambiente, e seus valores não podem ser alterados pelo sistema. Portanto, uma variável é considerada de entrada se e somente se ela aparecer apenas em condições. Do mesmo modo, uma variável é considerada de saída se e somente se ela aparecer apenas em ações. Timers podem aparecer tanto em condições como em ações. Para diferenciar os timers de outras variáveis, eles devem ter como sufixo a palavra "timer" quando forem especificados na CNL.

O algoritmo para identificar variáveis opera iterando sobre a lista de CF's, analisando as condições, que são compostas por uma conjunção de disjunções, e as ações.

Ao analisar condições, variáveis são extraídas do papel "Condition Patient (CPT)". Caso a variável ainda não tenha sido identificada, uma nova variável é criada. Por exemplo, a tabela 1 mostra que "the voltage" é o CPT da primeira condição. Nesse caso, uma variável "the_voltage" seria criada. Em seguida, verificamos se a variável tem a palavra "timer" como sufixo. Se isso ocorrer, a variável é classificada como um timer, caso não ela é classificada como uma entrada, dado que ela apareceu em uma condição.

Para inferir o tipo da variável, levamos em consideração o conteúdo do papel "Condition To Value (CTV)". No caso da variável "the_voltage", ela é associada com o valor 80 na primeira condição da Tabela 1, e portanto ela é considerada uma variável do tipo inteiro.

Na análise de ações, as variáveis são extraídas do papel "Patient (PAT)". O valor da variável corresponde ao conteúdo do papel "To Value (TOV)". Caso o verbo da ação seja "reset", o papel "To Value (TOV)" será vazio, e nesse caso o valor que será atribuído à variável será o tempo global do sistema, que pode ser um inteiro ou um número de ponto flutuante. No nosso caso, assumimos que o

tempo será sempre medido de maneira densa (usando o tipo de ponto flutuante), que é mais genérico e preciso do que a medição de maneira discreta (usando inteiros). Como o valor do tempo global do sistema ainda não é conhecido, atribuímos o valor 0.0 como possível valor para a variável para identificá-la como uma variável de ponto flutuante.

A inferência dos tipos das variáveis é feita analisando-se o valor associado a elas. Caso a variável seja de entrada ou saída, seu tipo pode ser um booleano, inteiro, número de ponto flutuante ou um tipo enumerável (para o caso de strings). O tipo enumerável não é suportado em um DFRS e, portanto, as variáveis com esse tipo são posteriormente mapeadas para inteiros.

As variáveis identificadas são separadas em três listas de acordo com o seu tipo (entrada, saída e timer).

O próximo passo é a identificação das funções. As funções caracterizam o comportamento do sistema. Elas são agrupadas de acordo com o Agente (AGT), ou seja, elas são atreladas ao componente que deve realizar as ações da função. Cada função corresponde a uma lista de ações que é mapeada para seus respectivos guardas (condições) discreto e temporal. O algoritmo, portanto, itera sobre os CF's para identificar os guardas e suas ações correspondentes.

As expressões de condição são conjunções de disjunções. Uma restrição que impomos é que as conjunções devem ter sempre guardas do mesmo tipo, ou seja, não podemos misturar guardas discretos e temporais dentro de uma mesma conjunção. Isso é necessário porque queremos dividir as condições em duas categorias distintas (discretas e temporais), como foi explicado na seção 1.2.

Depois de identificar os guardas discretos e temporais, o algoritmo itera sobre as ações do CF e cria uma lista de instruções de ação. Cada ação é representada como uma instrução de atribuição que altera o estado do sistema.

Com as funções e variáveis identificadas, falta apenas gerar os valores iniciais das variáveis. O algoritmo itera sobre as variáveis e atribui um valor para a variável de acordo com seu tipo. Caso a variável seja do tipo inteiro, ela recebe o valor 0, caso seja do tipo ponto flutuante recebe o valor 0.0, e caso seja do tipo booleano recebe o valor false.

Com isso, o DFRS é criado a partir de todos os elementos identificados.

3.2. Geração de CSP a partir de DFRS

A criação dos processos CSP que representam o DFRS foi dividida em 6 passos. Esses passos são listados a seguir e serão detalhados nas próximas subseções.

1. Criação dos processos que representam a comunicação entre processos (memória);
2. Criação do processo que representa a interação com o ambiente;
3. Criação dos processos que representam o comportamento do sistema;
4. Criação do processo que representa a passagem de tempo;
5. Criação dos processos que representam o comportamento cíclico do DFRS.

3.2.1. Representação da memória

Nesse passo são gerados dois processos (MCELL e MEMORY) que simulam uma memória onde os valores das variáveis de entrada e saída do sistema são armazenados. A definição da memória tem a seguinte forma geral:

```
1  datatype VAR = var1 | ...
2  datatype TYPE = B.Bool | I.INTEGER_VALUES
3  initialBinding = {(var1, B.false), …}
4  channel get, set : VAR.TYPE
5  MCELL (var,val) = get!var!val -> MCELL (var, val)
6          [] set!var?val' : range(tag(val)) -> MCELL(var, val')
7  MEMORY(binding) = ||| (var,val) : binding @ MCELL(var,val)
8  SYSTEM_MEMORY = MEMORY(initialBinding)
```

A linha 1 representa a definição das variáveis do sistema. A linha 2 representa a definição dos tipos dos possíveis valores. A linha 3 representa os valores iniciais das variáveis. A linha 4 representa os canais usados para manipular a memória. A memória, como mostrado nas linhas de 5 a 7 é representada por um conjunto de células de memória. Cada célula representa uma variável e tem duas operações possíveis, ler o valor da variável ou escrever um novo valor. A memória do sistema é inicializada de acordo com os valores iniciais definidos em *initialBinding*, como mostrado na linha 8. As funções range e tag usadas na linha 6 foram omitidas, mas elas representam, respectivamente, uma função que retorna os possíveis valores de um tipo de uma variável e uma função que retorna o tipo de um valor. Essas funções são usadas para garantir

que apenas valores compatíveis com o tipo da variável da célula de memória possam ser escritos na célula [3].

Para criar as variáveis e o *initialBinding*, apenas iteramos pelas variáveis identificadas no DFRS para, de acordo com seu tipo, declarar a variável e seu valor inicial.

3.2.2. Interação com o ambiente

A interação de um sistema com o ambiente é responsável pela atualização das entradas do sistema. Em CSP, essa comunicação é feita através de canais. Cada canal representa um sinal enviado ao sistema. Na declaração dos canais, é necessário explicitar o tipo dos valores que podem ser comunicados através deles. Esses valores representam os dados carregados pelos sinais enviados ao sistema.

O processo ENVIRONMENT_SYNC representa a interação com o ambiente. Esse processo tem a seguinte forma geral:

1 *channel* c_i *: CHANNEL_TYPE*
2 *channel memory_state : VAR.TYPE*
3 *ENVIRONMENT_SYNC =* c_i*?val -> set!*var_{ci}*!val ->*
4 *memory_state!* var_{ci}*!val -> SKIP*

Quando um valor é recebido do ambiente através de um canal, esse processo atualiza a posição de memória da variável que está relacionada com a entrada provida ao sistema. Depois disso, o processo comunica através do canal *memory_state* o valor que acabou de ser lido. A comunicação através do canal *memory_state* é utilizada pelo CSPTIO para checar a conformidade da implementação de acordo com sua especificação.

3.2.3. Comportamento do sistema

Nesse momento são criados os processos que representam o comportamento que o sistema deve ter. No DFRS, esse comportamento é capturado na identificação das funções. As funções são agrupadas de acordo com o componente que realiza as ações como foi visto na seção 3.1.3. Portanto, para cada componente do sistema é criado um processo que descreve o que o componente deve fazer. A estrutura geral do processo criado é a seguinte:

1 P_i = *get!*var_1*?I.*$varLocal_1$ → ... → *get!*var_n*?I.*$varLocal_n$→
2 *get!*$\eta_{i,1}$*?I.*$\eta Local_{i,1}$ → ... → *get!*$\eta_{i,k}$*?I.*$\eta Local_{i,k}$→
3 (*(if eval(*$\phi D_{i,1}$*)* ∧ $\eta Local_{i,1}$ *then set!disTrans!B.true* → $Act_{i,1}$→
4 $MS_{Act_{i,1}}$→*SKIP else STOP)*
5 □ … □
6 (*(if eval(*$\phi D_{i,k}$*)* ∧ $\eta Local_{i,k}$ *then set!disTrans!B.true* → $Act_{i,k}$→
7 $MS_{Act_{i,k}}$→*SKIP else STOP)*
8 □ *(if* ¬*(eval(*$\phi D_{i,1}$*)* ∧ $\eta Local_{i,1}$*)* ∧ ... ∧ ¬*(eval(*$\phi D_{i,k}$*)* ∧ $\eta Local_{i,k}$*) then SKIP*
9 *else STOP))*

Primeiramente, o processo lê todas as variáveis do sistema (var_1 ... var_n) e guarda seus valores em variáveis locais ($varLocal_1$ ... $varLocal_n$). O mesmo é feito para as variáveis $\eta_{i,k}$ (eta). As condições temporais são simbolicamente associadas às variáveis $\eta_{i,k}$. Uma variável $\eta_{i,k}$ recebe o valor *true* quando o tempo decorrido pode satisfazer a condição temporal associada a ela. Os valores coletados nas variáveis locais são utilizados para avaliar as condições das ações.

No processo, cada ação é descrita por meio de estruturas condicionais. Caso a condição seja satisfeita, o conjunto de atribuições às variáveis do sistema correspondente às ações é executado. Além disso, as atribuições são comunicadas através do canal *memory_state* ($MS_{Act_{i,k}}$) e a variável auxiliar *disTrans* recebe o valor *true*, indicando que uma transição discreta foi realizada. As diferentes ações que um componente executa são compostas no processo por meio do operador de escolha externa (□). Quando nenhuma condição é satisfeita (última escolha externa), o processo se comporta como SKIP.

3.2.4. Passagem de tempo

Como foi visto na seção anterior, as variáveis de tempo $\eta_{i,k}$ (eta) são tratadas apenas simbolicamente. Cada condição temporal gera uma variável $\eta_{i,k}$. Elas tratadas como variáveis booleanas que recebem o valor *true* apenas quando o tempo decorrido pode satisfazer a condição temporal associada a ela. Valores concretos para essas variáveis temporais só são atribuídos depois, com o auxílio de um *SMT solver*.

O processo TIME_PASSING tem a seguinte formal geral:

1 *TIME_PASSING = time_update_started* → *set!* $\eta_{1,1}$*!B.false* → … →
2 *set!* $\eta_{i,k}$*!B.false* → *get!*var_1*?I.*$varLocal_1$ → … →
3 *get!*var_n*?I.*$varLocal_n$ → (∀*i,k* : ∃$\phi T_{i,k}$ • *(if eval(*$\phi D_{i,k}$*) then*

4 *set!* $\eta_{i,k}$*!B.true* → *SKIP else STOP)*
5 □ *if* ∀*i,k* : ∃$\phi T_{i,k}$ ● ¬*eval*($\phi D_{i,k}$)) *then*ζ → *SKIP else STOP*
6 *; time_update_finished* → *SKIP*

Os eventos *time_update_started* e *time_update_finished* são usados para sincronizar esse processo com os outros. Como foi dito, para cada condição temporal, uma variável $\eta_{i,k}$ é gerada. No início desse processo, todas as variáveis $\eta_{i,k}$ recebem o valor *false*, e as variáveis do sistema são lidas e seus valores são guardados em variáveis locais. Depois disso, o processo analisa se o sistema está em um estado onde uma condição discreta ($\phi D_{i,k}$) é verdadeira e existe uma condição temporal ($\phi T_{i,k}$) correspondente. Se esse for o caso, isso significa que o tempo decorrido influencia se a transição ocorrerá ou não. Assim, o valor *true* é atribuído a $\eta_{i,k}$ para demonstrar que foi decorrido a quantidade de tempo necessária para que a condição seja verdadeira. Portanto, quando o valor de $\eta_{i,k}$ for avaliado em uma condição, ele será *true* e assim as ações correspondentes podem ocorrer. Valores concretos de tempo serão determinados posteriormente com o auxílio de um *solver*. Porém esses valores terão que respeitar as restrições associadas a $\eta_{i,k}$.

Caso o sistema esteja em um estado onde a quantidade de tempo decorrido já não influencia, então qualquer quantidade de tempo pode decorrer sem produzir nenhuma saída. Esse comportamento é capturado pela última escolha externa no processo. Caso nenhuma condição temporal seja satisfeita, o processo realiza o evento ζ, sem atribuir o valor *true* a alguma variável $\eta_{i,k}$. Desse modo, fica explícito que o sistema atingiu um estado onde o tempo não é mais relevante.

3.2.5. Comportamento Cíclico

Um DFRS interage de forma cíclica com o ambiente. Neste passo, criamos um processo para representar este comportamento cíclico do sistema. O processo SPECIFICATION tem a seguinte forma geral:

1 *SPECIFICATION = ((DISCRETE_TRANS ; ENVIRONMENT_SYNC;*
2 *time_update_started → time_update_finished → SKIP)* ||
3 *TIME_PASSING) ; SPECIFICATION*

Primeiramente, o processo DISCRETE_TRANS permanece em um loop enquanto transições discretas puderem ser realizadas. Nesse momento, o processo termina com sucesso, e então o processo ENVIRONMENT_SYNC é executado. O sistema interage com o ambiente para mudar o seu estado. O

tempo decorrido desde a última interação com o ambiente é determinado pelo processo TIME_PASSING, que é composto em paralelo (||), e usa os eventos *time_update_started* e *time_update_finished* para sincronizar. Finalmente, o processo SPECIFICATION se comporta como ele mesmo, ou seja, ele volta ao início para ser executado mais uma vez.

O processo DISCRETE_TRANS é definido da seguinte maneira:

```
1  DISCRETE_TRANS  = set!disTrans!B.false → (P_1 ||| ... ||| P_i);
2          get!disTrans!B.engaged →
3          (if engaged then INSPECT_MEM ; get!memUpdate?B.changed →
4                  (if changed then UPDATE_MEM ; DISCRETE_TRANS
5                  else SKIP)
6          else SKIP )
```

Primeiramente, o processo atribui o valor *false* à *disTrans*. Depois, o sistema se comporta como uma composição dos processos que representam o comportamento do sistema ($P_1 ||| ... ||| P_i$). Quando todos os processos terminam, a variável *disTrans* é avaliada para verificar se houve alguma transição discreta em algum dos processos P_i. Porém, para permanecer no loop de DISCRETE_TRANS é necessário que as transições discretas que foram realizadas tenham algum efeito colateral, ou seja, que elas alterem a memória do sistema. O processo INSPECT_MEM realiza esta verificação para constatar se houve mudança na memória. Caso tenha algum efeito colateral, a variável *memUpdate* recebe o valor *true* e em seguida as mudanças são aplicadas à memória através do processo UPDATE_MEM.

O processo SYSTEM define o sistema e amarra o funcionamento de todos os outros processos. Ele é definido como uma composição paralela do processo SPECIFICATION com o processo MEMORY. Os dois processos são sincronizados através dos eventos de *get* e *set*. O processo SYSTEM tem a seguinte forma:

$$1 \quad SYSTEM = SPECIFICATION \underset{\{|get, set|\}}{||} MEMORY(initialBinding)$$

4. Verificação de Conformidade CSPTIO

As expressões para verificação de conformidade CSPTIO são detalhadas em [3]. Para realizar a verificação, utilizamos a ferramenta FDR [6]. FDR é uma ferramenta para checagem de modelos. Para verificar se o processo CSP que representa o sistema na implementação está conforme com o processo CSP que representa o sistema na especificação, usamos a noção de refinamento. Um processo Q refina um processo P se ele satisfaz pelo menos as mesmas condições que P, ou seja, o processo Q deve fazer, pelo menos, tudo o que o processo P é capaz de fazer.

Utilizamos, particularmente, o refinamento através de *traces*. Esse tipo de refinamento é baseado na sequência de eventos que um processo realiza (*traces* do processo). Portanto, um processo Q refina um outro processo P, através de *traces,* se todas as sequências possíveis de comunicação que Q consegue executar também são possíveis em P ($traces(Q) \subseteq traces(P)$). Esse relacionamento é escrito como $P \sqsubseteq_T Q$.

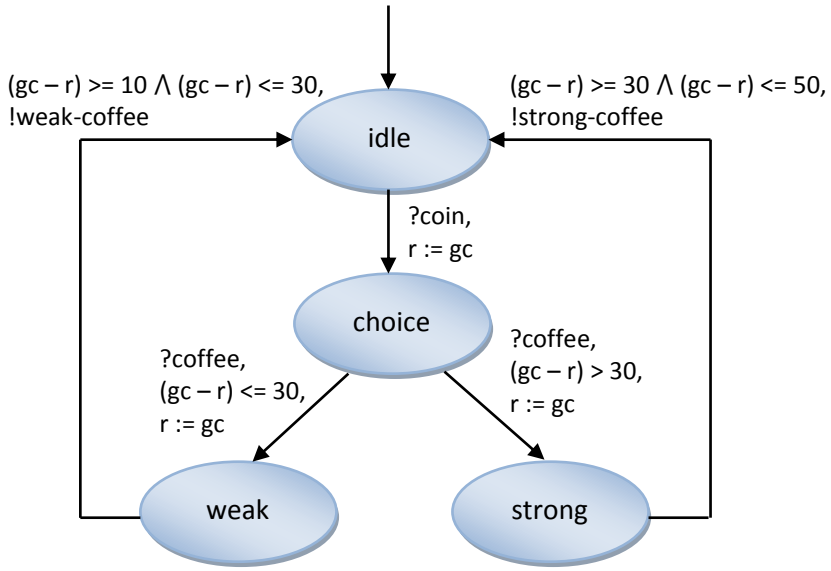
A verificação de sentenças em FDR é feita por meio de assertivas. Em nosso caso, criamos três assertivas, uma para verificar se a implementação refina a especificação, outra para verificar se a especificação refina a implementação, e outra para verificar se a implementação está CSPTIO (relação de conformidade temporal) com a especificação.

Para verificar a conformidade, verificamos as três assertivas explicadas no parágrafo anterior. Caso todas elas sejam verdadeiras, pode ainda haver um problema temporal na implementação. Nesse caso, é preciso gerar os contra-exemplos, usando o FDR, e para cada contra-exemplo é derivado um problema no Z3 [7], que pode encontrar possíveis erros temporais. A integração com o FDR para verificação de conformidade foi implementada, assim como também foi implementada a geração dos contra-exemplos. Porém, a integração com o Z3 não foi realizada.

A comunicação com o FDR foi feita de forma automática com o auxílio da ferramenta ATG, desenvolvida pelo aluno de doutorado Sidney Nogueira.

5. Estudo de Caso

Utilizamos como estudo de caso um exemplo de uma máquina de café. Este exemplo é uma adaptação da máquina de café apresentada em [5]. O funcionamento da máquina é apresentado no diagrama da figura 1.



Inicialmente, a máquina se encontra no estado *idle*. Quando uma moeda é inserida (*?coin* representa a entrada de uma moeda), o sistema vai para o estado *choice* e reseta o timer de requisições (*r* na figura). Isso é equivalente a atribuir o tempo global atual (*gc*) à variável *r*. Após inserir uma moeda e selecionar a opção de café, o sistema vai para um dos dois estados: *weak* ou *strong*. Caso o usuário selecione a opção de café (*?coffee*) dentro de 30 segundos, o sistema vai para o estado *weak*, e um café fraco é preparado. Caso contrário, o sistema vai para o estado *strong*, e um café forte é preparado.

O sistema leva tempos diferentes para preparar cada tipo de café. Um café fraco é entregue ao usuário de 10 a 30 segundos depois do pedido, enquanto que um café forte é entregue de 30 a 50 segundos depois que o pedido é feito. Por esse motivo, o timer de requisições sofre um reset quando a escolha é feita.

A lista de requisitos para a máquina de café é apresentada a seguir.

- When the system mode is idle, and the coin sensor changes to true, the coffee machine system shall: reset the request timer, assign choice to the system mode.
- When the system mode is choice, and the coffee request button changes to pressed, and the request timer is lower than or equal to 30, the coffee machine system shall: reset the request timer, assign preparing weak coffee to the system mode.

- When the system mode is choice, and the coffee request button changes to pressed, and the request timer is higher than 30, the coffee machine system shall: reset the request timer, assign preparing strong coffee to the system mode.
- When the system mode is preparing weak coffee, and the request timer is greater than or equal to 10, and the request timer is lower than or equal to 30, the coffee machine system shall: assign idle to the system mode, assign weak to coffee machine output.
- When the system mode is preparing strong coffee, and the request timer is greater than or equal to 30, and the request timer is lower than or equal to 50, the coffee machine system shall: assign idle to the system mode, assign strong to coffee machine output.
- When the system mode changes to idle, the coffee machine system shall assign undefined to coffee machine output.

Os requisitos foram utilizados como entrada na ferramenta NAT2TEST para geração dos *Case Frames*. Os *Case Frames* foram usados como entrada para a ferramenta implementada neste trabalho, para a geração do código CSP. Foram gerados dois códigos CSP a partir do mesmo conjunto de *Case Frames*, um que representa a especificação do sistema e outro que representa a implementação. A ideia é que futuramente seja provido um modelo da especificação e um modelo da implementação para que haja a verificação de conformidade.

A geração dos códigos levou em média 200 ms. Os códigos gerados são submetidos ao FDR, de maneira automática, para verificação de conformidade CSPTIO. A tabela 2 dá alguns resultados para a verificação CSPTIO dos dois arquivos.

Tabela 2 - Resultados para verificação de CSPTIO

Métrica	Valor
Linhas de código (CSP)	120
Variáveis	15
Estados (±)	7K
Transições (±)	7K

A geração dos contra-exemplos, usando o FDR, foi feita para diversos valores de *k*, onde *k* representa o número de passagens de tempo que o sistema faz. Se *k = 1*, por exemplo, o sistema estará inicialmente no estado *idle* e serão enumerados os *traces* (sequência de eventos) que envolvam apenas uma passagem de tempo. Nesse caso, os *traces* corresponderão ao momento em que a

moeda deve ser inserida. A tabela 3 dá mais detalhes sobre a geração dos contra-exemplos.

Tabela 3 - Resultados da geração de traces usando o FDR

	K = 1	**K = 2**	**K = 3**
Nº de traces	4	22	134
Tempo médio de geração (s)	5	8	13

6. Conclusões e Trabalhos Futuros

A contribuição deste trabalho foi de criar um módulo de verificação de conformidade para a ferramenta NAT2TEST. No momento, é possível gerar modelos em CSP da especificação e da implementação de um sistema através dos requisitos expressos em linguagem natural. Com os modelos, podemos fazer a verificação de conformidade e também enumerar *traces* que serão encaminhados a um *solver* (Z3), que poderá encontrar possíveis erros temporais no sistema.

Possíveis melhorias na ferramenta seriam:

- Integração com Z3;
- Criar uma interface gráfica;

7. Referências Bibliográficas

[1] Carvalho, G., Barros, F., Lapschies, F., Schulze, U., Peleska, J.: Model Based Testing from Controlled Natural Language Requirements. 18th International Workshop on Formal Methods for Industrial Critical Systems, 2013, Madrid, Spain.

[2] Allen, J.: Natural Language Understanding. Benjamin/Cummings (1995)

[3] Carvalho, G., Sampaio, A., Mota, A.: A CSP Timed Input-Output Relation and a Strategy for Mechanised Conformance Verification. 15th International Conference on Formal Engineering Methods, 2013, Queenstown, New Zealand.

[4] J. Peleska, E. Vorobev, F. Lapschies, and C. Zahlten. Automated modelbased testing with RT-Tester. Technical report, Universität Bremen, 2011.

[5] Piel, É., González-Sanchez, A., Groβ, H.G.: Built-in data-flow integration testing in large-scale component-based systems. In: Proceedings of ICTSS. pp. 79-94 (2010)

[6] FDR – Department of Computer Sciente – University of Oxford. Disponível em: http://www.cs.ox.ac.uk/projects/concurrency-tools/

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

[8] Larsen, K., Mikucionis, M., Nielsen, B.: Online Testing of Real-time Systems using Upaal: Status and Future Work. In: Dagstuhl Seminar Proceedings volume 04371: Perspectives of Model-Based Testing (2004)

[9] Schneider, S.A. (1999). Concurrent and Real Time Systems: the CSP Approach, John Wiley.

[10] Hong-Viet Luong , Thomas Lambolais , Anne-Lise Courbis, Implementation of the Conformance Relation for Incremental Development of Behavioural Models, Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems, September 28-October 03, 2008, Toulouse, France.

8. Assinaturas

_____

Augusto Cezar Alves Sampaio
Orientador

_____

Eduardo Bastos Rocha
Aluno