UM MAPEAMENTO ENTRE ASSEMBLY E CSP: ANALISANDO FORMALMENTE PROGRAMAS CONCORRENTES

Trabalho de Conclusão de Curso

Engenharia da Computação

Rafael Farias Cabral

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

UNIVERSIDADE
DE PERNAMBUCO

Rafael Farias Cabral

UM MAPEAMENTO ENTRE ASSEMBLY E CSP: ANALISANDO FORMALMENTE PROGRAMAS CONCORRENTES

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

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

Universidade de Pernambuco
Escola Politécnica de Pernambuco
Graduação em Engenharia de Computação

Recife - PE, Brasil

23 de maio de 2012

ESCOLA POLITÉCNICA
DE PERNAMBUCO

MONOGRAFIA DE FINAL DE CURSO

Avaliação Final (para o presidente da banca)*

No dia 29 de 5 de 2012, às 15:00 horas, reuniu-se para deliberar a defesa da monografia de conclusão de curso do discente RAFAEL FARIAS CABRAL, orientado pelo professor Gustavo Henrique Porto de Carvalho, sob título Um Mapeamento entre Assembly e CSP: Analisando Formalmente Programas Concorrentes, a banca composta pelos professores:

Joabe Bezerra de Jesus Júnior

Gustavo Henrique Porto de Carvalho

Após a apresentação da monografia e discussão entre os membros da Banca, a mesma foi considerada:

☒ Aprovada ☐ Aprovada com Restrições* ☐ Reprovada

e foi-lhe atribuída nota: 9,5 (nove e meio)

*(Obrigatório o preenchimento do campo abaixo com comentários para o autor)

O discente terá 7 dias para entrega da versão final da monografia a contar da data deste documento.

Joabe Bezerra de Jesus Júnior

JOABE BEZERRA DE JESUS JÚNIOR

Gustavo Henrique

GUSTAVO HENRIQUE PORTO DE CARVALHO

* Este documento deverá ser encadernado juntamente com a monografia em versão final.

Agradecimentos

Agradeço primeiramente aos meus pais, Maurício e Valéria Cabral, pela educação e pelas condições que me deram para alcançar mais este objetivo. Também aos meus irmãos, Amanda e Tiago, por me acompanharem e para os quais espero servir de exemplo em suas empreitadas.

À minha psicóloga e amiga, Maria Luiza, pois foi com seu apoio e conselhos que consegui chegar até aqui.

Agradeço também aos meus colegas de turma pela amizade e pelo alto nível intelectual que proporcionaram ao longo de todos os períodos da graduação; me dando motivos para buscar melhorar sempre.

Um agradecimento especial ao meu professor orientador, Gustavo Carvalho, pela grande atenção e apoio prestados.

Por fim, agradeço a Deus por estar sempre presente em minha vida, me ajudando a acreditar na minha capacidade e perseverança.

Resumo

A porcentagem de funcionalidades fornecidas através de *software* pelos mais diversos dispositivos é cada vez maior. Essa tendência tem tornado os *softwares* maiores e mais complexos. Em contrapartida, a maior quantidade (de linhas) de código gera um aumento na probabilidade de falhas sendo propagadas nas várias fases do desenvolvimento. É comum ainda que parte dessas funcionalidades seja implementada de maneira concorrente, seja visando melhor desempenho ou pela necessidade de interação com o usuário ou com outro sistema, por exemplo. O problema é que a habilidade humana e os testes de *software* mais comuns são normalmente insuficientes para verificar a presença de falhas de concorrência. Tendo em vista essa dificuldade de verificação, este trabalho propõe uma tradução direta do código de máquina para especificações CSP (*Communicating Sequential Processes*), permitindo a análise formal de código dessa natureza. Foram definidos dois conjuntos de regras de mapeamento: gerais e de instrução. As regras foram, então, aplicadas sobre o código x86 de dois estudos de caso escritos em C. Em seguida, a análise com a ferramenta PAT (*Process Analysis Toolkit*), que utiliza o dialeto CSP# e permite que sejam feitas verificações da especificação, tais como *deadlock-free* e asserções LTL (*Linear Temporal Logic*) obteve resultados satisfatórios, identificando corretamente os problemas conhecidamente nos sistemas verificados.

Palavras-chave: CSP, verificação formal, Assembly, *deadlock*.

Abstract

The percentage of functionalities provided by software for various devices is increasing. Due to this tremb, software has also become something bigger and more complex. However, more (lines of) code produce a higher probability of faults being propagated in the different stages of development. It is also common that some of these functionalities get implemented concurrently, either to improve performance or to provide clean interaction with the user or another system, for example. The problem is that human skills and most common software testing technics are usually not sufficient to verify the presence of concurrency failures. Given this gap in the verification, this paper proposes a direct translation of the machine code into CSP (Communicating Sequential Processes) specifications, enabling the formal analysis of such code. Two sets of mapping rules were defined: general rules and instruction rules. The rules were then applied on the x86 codes from two case studies written in C. After that, the formal analysis was performed with the PAT (Process Analysis Toolkit), which uses the dialect CSP# and allows some assertions about the specification, like deadlock-free and LTL (Linear Time Logic) assertions obtained satisfactory results, correctly identifying the problems known to exist in the verified systems.
Keywords: CSP, formal verification, Assembly, deadlock.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

API	*Application Programming Interface*
CS	*Code Segment*
CSP	*Communicating Sequential Processes*
CSP_M	*Machine-Readable CSP*
CSP#	*CSP Sharp*
DS	*Data Segment*
EAX	*Extended Accumulator Register*
EBP	*Extended Base Pointer*
EBX	*Extended Base Register*
ECU	*Electronic Control Units*
ECX	*Extended Count Register*
EDI	*Extended Destination Index*
EDX	*Extended Data Register*
EIP	*Extended Instruction Pointer*
ES	*Extra Segment*
ESI	*Extended Source Index*
ESP	*Extended Stack Pointer*
FDR	*Failures-Divergences Refinement Checker*
GCC	*GNU Compiler Collection*
JIT	*Just In Time Compiler*
JVM	*Java Virtual Machine*
LLVM	*Low Level Virtual Machine*
LLVM2CSP	*LLVM to CSP*
LTL	*Linear Temporal Logic*
MSDN	*Microsoft Development Network*
PAT	*Process Analysis Toolkit*
PC	*Program Counter*
ProBE	*Process Behaviour Explorer*
RCE	*Regras de Chamadas Externas*
RI	*Regras de Instruções*
RISC	*Reduced Instruction Set Computer*
SMT	*Satisfiability Modulo Theories*
SS	*Stack Segment*
VCC	*Verifying C Compiler*

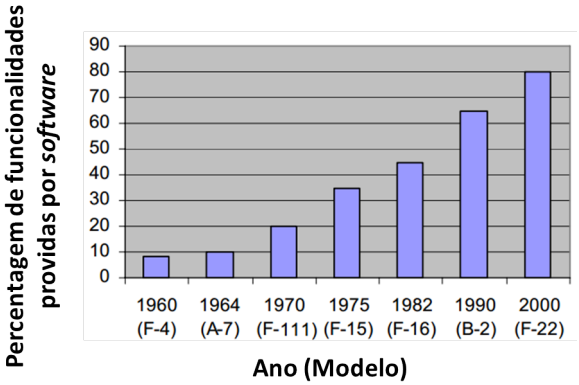
1 Introdução

A dependência por *software* nos mais diversos dispositivos computacionais encontrados no mercado vem crescendo notadamente. A miniaturização de componentes eletrônicos permitiu embuti-los nos dispositivos em maiores quantidades, fornecendo maior flexibilidade para o desenvolvimento de funcionalidades antes impensadas ou impraticáveis devido à complexidade para construção diretamente em *hardware*.

Carros, por exemplo, carregam consigo dezenas de ECU's (*Electronic Control Units*), que são unidades eletrônicas de controle, responsáveis por diversas funcionalidades oferecidas através de *software*. Tais unidades estão presentes até nos modelos mais simples e algumas podem ainda trabalhar em rede (CHARETTE, 2009).

O cenário se repete em outras áreas. As aeronaves militares, por exemplo, apresentaram um notável crescimento no percentual de funcionalidades providas por *software* para seus pilotos nas últimas décadas. Como mostra a Figura 1, o caça F-4, no ano de 1996, dispunha menos de 10% delas através de *software*. Já no F-22, em 2000, esse número alcançava os 80% (DVORAK; LYU, 1996).

Figura 1: Crescimento de funcionalidades providas por *software*
[Fonte: Adaptado de Dvorak e Lyu (1996)]



Entretanto, prover mais funcionalidades através de *software* implica em um aumento da quantidade de linhas de código. Essas, por sua vez, ocasionam acúmulo e propaga-

ção de falhas conforme se avança nas várias fases do desenvolvimento: levantamento de requisitos, modelagem, implementação e testes. Estatisticamente, em um processo de desenvolvimento bem controlado de larga escala, é comum assumir que haverá de 0,1 a 1 defeito residual para cada mil linhas efetivas de código, excluindo comentários e linhas em branco (DVORAK; LYU, 1996). Logo, principalmente quando aplicado a sistemas críticos, como é o caso dos freios ou dos *airbags* de um carro, é bastante importante que essa taxa seja reduzida ao mínimo possível.

1.1 Qualificação do Problema

Diferente dos métodos formais, os modelos utilizados na engenharia de *software* convencional não fornecem uma maneira sistemática de especificar, desenvolver e verificar artefatos de *software* (WING, 1990).

No que diz respeito à verificação, mesmo passando por testes de caixa-branca, testes estruturais que buscam problemas lógicos diretamente no código fonte e fluxos gerados por ele, é possível que outros problemas, como os associados a condições de corrida (WING, 1990), passem despercebidos. Isso acontece por que esse tipo de teste, *a priori*, não considera as decisões de escalonamento feitas pelo processador.

Aquela, porém, é apenas a maneira convencional de fazer a verificação de *software*. Outra maneira é a utilização de métodos formais. Não tão difundida pela dificuldade de aplicação em sistemas grandes e complexos; e pela baixa oferta de mão-de-obra especializada (WING, 1990).

Um método é formal se tem uma sólida base matemática, tipicamente provida por uma linguagem de especificação formal. Com ele é possível avaliar ambiguidade, completude e consistência de um sistema – não necessariamente computacional. Além disso, devido a essa base, fornece meios de provar a corretude do sistema modelado (WING, 1990).

Em particular, a linguagem CSP (*Communicating Sequential Processes*) foi concebida para descrever sistemas de componentes que interagem entre si. Tais componentes, ou processos, são considerados de forma independente e possuem interfaces, pelas quais eles interagem com o ambiente (SCHNEIDER, 1999). Através de CSP, é possível descrever todas as combinações de processos sendo executados sequencialmente, paralelamente ou concorrentemente. Contudo, a adoção de linguagens formais tem certa resistência; uma vez que as equipes de desenvolvimento, normalmente, não estão familiarizadas com seus aspectos.

A descrição formal de modelos CSP passível de processamento computacional é possibilitada por dialetos que utilizam caracteres apropriados e notações um pouco diferentes da notação tradicional, como CSP#, que é processado, simulado e verificado através da ferramenta PAT (*Process Analysis Toolkit*).

Alguns autores já estudaram e criaram meios de atacar esse problema de verificação de sistemas concorrentes. Lima (2011) propôs um mapeamento para análise de códigos concorrentes em Java. Em seu trabalho, o mapeamento se dava de instruções do *bytecode* Java para CSP#, pois o escalonamento de *threads*, feito pela JVM (Java *Virtual Machine*), se dá neste nível. Porém, em algum ponto da execução, o *bytecode* será traduzido para código de máquina nativo, pois é o que processador entende. Logo, o mapeamento de instruções tornaria o mapeamento mais abrangente.

Já a solução de Kleine et al. (2011), parte de códigos de baixo nível escritos para uma máquina virtual especializada, transformando-os em código CSP_M(*Machine ReadableCSP*).

Entretanto, a abordagem com álgebra de processos não é a única possível. Maus, Moskal e Schulte (2008) propuseram a verificação de *assembler* x86 decorado através de traduções subsequentes até chegar a um modelo de verificação baseado em um conjunto de fórmulas de primeira ordem e uso de um solucionador SMT (*Satisfiability Modulo Theories*). Porém, quando mal especificadas, suas verificações podem ser tarefas indecidíveis, ou seja, nunca terminam.

Então, o problema de pesquisa deste trabalho é: **como gerar automaticamente especificações CSP a partir de código *Assembly*?**

1.2 Objetivos

O objetivo principal deste trabalho é propor mapeamentos de código *Assembly* para um modelo formal CSP que simule o funcionamento do sistema computacional visando automatizar a verificação da presença, ou ausência, de falhas decorrentes do uso da programação concorrente oferecida pelo sistema operacional.

1.2.1 Objetivos Específicos

- Definir regras de mapeamento gerais, que modelarão o funcionamento do sistema computacional e características de concorrência;
- Definir regras de nomenclatura, que atribuirão nomes aos elementos do código tra-

duzidos para CSP;

- Definir regras de mapeamento das instruções, que transformarão o *Assembly* em elementos CSP equivalentes;
- Analisar utilizando PAT a ausência de *deadlocks* e a não-terminação.

1.3 Resultados e Impactos Esperados

O resultado esperado deste trabalho é um conjunto de regras de mapeamento de código *Assembly* para uma especificação formal equivalente em CSP#.

Espera-se que a transformação, e posterior verificação formal, obtida pela aplicação de tais regras, forneça aos desenvolvedores de *software* uma maneira efetiva de reduzir a quantidade de falhas de um programa antes de sua implantação em ambiente de produção. Por ser automatizado e não exigir muita profundidade de conhecimento acerca de métodos formais, também é esperado que a resistência atribuída à utilização desta solução seja menor do que a de outras abordagens.

1.4 Estrutura da Monografia

Além deste capítulo, o trabalho está dividido em mais quatro capítulos:

- Capítulo 2 - Referêncial Teórico: descreve conceitos usados como base para o desenvolvimento do trabalho. Contém informações sobre os fundamentos de CSP, suas ferramentas e dialetos, lógica temporal, *Assembly*, ambiente de execução e programação concorrente. Além disso, discute soluções diferentes em trabalhos escritos por outros autores.
- Capítulo 3 - Método de Pesquisa: expõe de que maneira foi realizado o estudo, a definição dos exemplos, a extração das regras de mapeamento e a análise dos resultados.
- Capítulo 4 - Resultados: apresenta o conjunto de regras para tradução de código x86 *assembler* em comandos CSP# e a aplicação do mesmo nos exemplos definidos.
- Capítulo 5 - Considerações Finais: analisa o impacto dos resultados obtidos e cita possíveis trabalhos futuros.

2 Referencial Teórico

Neste capítulo é apresentada a base teórica necessária ao entendimento de seções posteriores deste trabalho. Aqui são discutidos os fundamentos de CSP, as particularidades de CSP_M e de CSP# , o *framework* PAT, alguns conceitos de concorrência, o código de máquina e a manipulação de *threads* em C no ambiente Windows.

2.1 Conceitos Chaves

2.1.1 CSP

A linguagem formal CSP impõe uma maneira de analisar o mundo através da especificação de composições e interações entre sistemas independentes. Esse nível de abstração permite que CSP consiga representar não apenas sistemas computacionais, como qualquer sistema de uma maneira genérica. Por exemplo, podem-se considerar os departamentos de uma empresa como subsistemas independentes onde haja certa necessidade de comunicação entre eles.

O propósito final de CSP é prover uma visão diferente para a análise e a especificação das diversas possibilidades de interação entre esses componentes (SCHNEIDER, 1999). Sendo o processo e o evento os dois mais básicos deles. O primeiro pode ser pensado como uma caixa preta que fornece interfaces de entrada e de saída. Por exemplo, considerando uma cafeteira simples que disponibiliza dois tamanhos de café como um processo, tem-se os botões “Longo” e “Curto” como interfaces de entrada e a torneira como a de saída. Nesse caso, o processo “Cafeteira” omite processos internos a ele como o aquecimento da água e a coagem do café.

Já um evento descreve uma ação atômica dentro de um processo, uma transição entre seus estados. A interface de um processo é descrita por um conjunto desses eventos. Portanto, no exemplo da cafeteira a interface pode conter os seguintes eventos: apertar botão “Curto”; apertar botão “Longo”; despejar café longo; e despejar café curto.

2.1.1.1 Transições

A semântica operacional de CSP define como seu interpretador deve executar as devidas transições entre processos. Por exemplo, tomando um processo definido por $P = a \rightarrow P'$, a única ação possível inicialmente é o acionamento do evento a. Essa execução pode ser descrita por $(a \rightarrow P') \xrightarrow{a} P'$. O próximo passo P', então, seguiria o mesmo pensamento a partir do processo resultante até que a execução termine.

2.1.1.2 Términos

Existem dois processos especiais que indicam o término da execução de um dado processo. Um deles é $STOP$, cujo conjunto de eventos de transição é vazio, ou seja, quando se chega a ele, não é possível fazer nenhum progresso na execução. O outro é denominado $SKIP$, que indica uma terminação com sucesso. Este, é executado gerando o evento $\surd$, isto é, $SKIP \xrightarrow{\surd} STOP$.

2.1.1.3 Pré-fixo

O conjunto de eventos externos de uma especificação em CSP é denotado por Σ. Dado um processo P, se $a \in \Sigma$ então pode-se escrever um processo em que $a \rightarrow P$ (leia-se a então P). Como a é o único evento habilitado para tal processo, sua execução é descrita por: $(a \rightarrow P) \xrightarrow{a} P$.

Considerando o caso de uma máquina copiadora que funcione apenas uma vez primeiramente digitalizando o documento para depois imprimi-lo, tem-se o processo descrito em (2.1), cuja interpretação passa por (2.2) e (2.3). Inicialmente (2.2), apenas o evento *digitalizar* está habilitado. Num segundo momento (2.3), o único evento habilitado é *imprimir*.

$$COPIADORA = digitalizar \rightarrow imprimir \rightarrow STOP \tag{2.1}$$

$$(digitalizar \rightarrow (imprimir \rightarrow STOP)) \xrightarrow{digitalizar} (imprimir \rightarrow STOP) \tag{2.2}$$

$$(imprimir \rightarrow STOP) \xrightarrow{imprimir} STOP \tag{2.3}$$

2.1.1.4 Recursão

A recursão em CSP é usada para descrever processos que podem rodar indefinidamente. Por exemplo, um abajur que é ligado diretamente à tomada apresenta apenas dois

possíveis estados. Sua execução – que passa por (2.5) e (2.6), depois volta a ser (2.4) – nunca termina.

$$ABAJUR = ligar \rightarrow desligar \rightarrow ABAJUR \tag{2.4}$$

$$(ligar \rightarrow (desligar \rightarrow ABAJUR)) \stackrel{ligar}{\longrightarrow} (desligar \rightarrow ABAJUR)) \tag{2.5}$$

$$(desligar \rightarrow ABAJUR)) \stackrel{desligar}{\longrightarrow} ABAJUR \tag{2.6}$$

2.1.1.5 Eventos compostos

Mesmo sendo fenômenos atômicos, os eventos de CSP são capazes de carregar pedaços de informação se utilizando de certa estrutura. Existem dois tipos de construção que resultam na composição de eventos. O mais simples deles é unir um evento simples com uma informação através do operador ".", por exemplo, *escolher.longo* e *escolher.curto* no caso da cafeteira.

Outra maneira é utilizando canais de comunicação. É possível definir um conjunto de valores possíveis na leitura e escrita de um canal declarando explicitamente esse tipo. Por exemplo, se *escolher* fosse especificado como um canal, cujo tipo fosse $T = \{longo, curto\}$, então o conjunto $\{escolher.t | t \in T\}$ seria o conjunto de eventos associado a este canal (SCHNEIDER, 1999).

Não há limite teórico para a quantidade de vezes que um evento pode ser composto pelo operador de composição, mas há situações em que a quantidade de configurações possíveis impactam na quantidade de estados possíveis e, portanto, na verificação do modelo; já que se trata de um produto cartesiano entre os tipos compostos.

2.1.1.6 Entrada e saída em canais

A escrita e leitura de valores nos canais é feita, respectivamente, com os operadores "!" e "?". Por exemplo, um $BUFFER$ de apenas uma posição pode dispor dos canais *entrada* e *saída*, ambos do tipo $ALCANCE = \{0, 1, 2\}$ descrito por (2.7). Depois de recebido como entrada, o valor v é escrito no canal *saída*, disponível para leitura.

$$BUFFER = entrada?v \rightarrow saída!v \rightarrow BUFFER \tag{2.7}$$

2.1.1.7 Escolha

Há duas maneiras de inserir uma escolha entre processos que desencadeiam diferentes fluxos de execução. São elas a escolha externa e a escolha interna. A primeira, denotada por $P_1 \Box P_2$ espera que o ambiente, através do próximo evento executado, decida qual dos processos tomará o controle. Há mais de um evento na interface dessa escolha, porém todos eles exclusivamente pertencentes à interface de um dos processos.

Já a segunda, denotada por $P_1 \sqcap P_2$, toma essa decisão internamente ao processo e sem a influência do ambiente. Ao entrar nele, uma transição silenciosa acontece decidindo qual entre os processos seguirá sua execução. Portanto, enquanto o primeiro tipo de escolha é determinístico, a escolha interna é não-determinística.

2.1.1.8 Alfabeto

O alfabeto de um processo P_1, denotado por $\alpha(P_1)$, é o conjunto de todos os eventos descritos por esse processo direta ou indiretamente, através das possíveis combinações dos canais de comunicação. Utilizando o exemplo do $BUFFER$, escreve-se:

$$\alpha(BUFFER) = \{entrada.0, entrada.1, entrada.2, saída.0, saída.1, saída.2\} \tag{2.8}$$

2.1.1.9 Paralelismo

Existem dois operadores em CSP que permitem descrever o comportamento paralelo de processos. Um deles é o paralelismo alfabetizado, ou com alfabetos, cujo operador é "||". Nesse tipo de paralelismo é necessário especificar como os processos irão interagir. Isso é feito fornecendo a interface dos processos como em (2.9).

$$P_{1\{a,b,c\}} ||_{\{a,b\}} P_2 \tag{2.9}$$

Sempre que se chegar a um ponto da execução onde haja um evento que pertença a ambas as interfaces listadas no operador, tal evento só será habilitado se os dois processos o têm habilitado simultaneamente. A execução desse evento (2.10) faz ambos os processos prosseguirem.

$$(a \rightarrow P'_1)_{\{a,b,c,\}} ||_{\{a,b\}} (a \rightarrow P'_2) \xrightarrow{a} P'_{1\{a,b,c\}} ||_{\{a,b\}} P'_2 \tag{2.10}$$

O segundo operador de paralelismo é chamado *interleaving*, simbolizado por "|||". Ao contrário do anterior, este operador permite que os processos executem de maneira

completamente independente uns dos outros (ROSCOE; HOARE; BIRD, 1997), incluindo casos onde os eventos dos processos possuam o mesmo nome, pois o evento será executado por apenas um dos deles. Assim, dado que $P_1 \xrightarrow{a} P_1'$, é possível ober:

$$P_1|||P_1 \xrightarrow{a} P_1'|||P_1 \tag{2.11}$$

2.1.1.10 Propriedades

Normalmente, depois de elaborada uma especificação CSP, deseja-se fazer algumas verificações acerca de sua corretude.

A ferramenta PAT oferece a verificação das seguintes asserções sobre o LTS (*Labeled Transition System* – representação interna dos modelos de entrada durante execução e verificação) de uma especificação CSP# (SUN; LIU; DONG, 2011):

- *Deadlock-freedom* (Ausência de *deadlock*): Verifica se não existem estados que não tenham transições possíveis – excetuando-se o estado de terminação com sucesso.
- *Divergence-freedom* ou *Livelock-freedom* (Ausência de *livelocks*): Verifica se um processo não pode realizar transições internas – transições que não são visíveis – para sempre, sem exercer nenhuma transição externa. Deve-se notar que esta é uma noção formal e específica de *livelock*.
- *Deterministic* (Determinístico): Checa se não existe algum estado em que um mesmo evento (dois ou mais eventos de mesmo nome) pode levar a dois estados diferentes.
- *Nonterminating* (Interminável): Verifica se não existe algum estado de terminação.
- *Reachability* (Alcançável): Visita os estados a procura de um em que o objetivo desejado é satisfeito.

2.1.2 FDR e CSP$_M$

FDR (*Failures Divergences Refinement checker*) é uma ferramenta capaz de fazer análise automática de processos CSP. É também utilizada na verificação de parte das propriedades citadas anteriormente (SCHNEIDER, 1999). Porém, sua função principal é de examinar se um processo CSP refina ou não outro. O refinamento permite saber se um processo reflete as propriedades de outro sendo que escrito de forma diferente.

Apesar das combinações entre processos CSP ter sido mostrada até aqui através de seus operadores algébricos, a automatização de sua análise exige que a descrição do modelo se pareça mais com uma linguagem de programação a fim de ser entendida pelo computador. Além dos operadores precisarem de uma versão textual, a estruturação do conteúdo do arquivo, a declaração de canais e os tipos de dados passados por eles também devem ser padronizadas (FSE, 2010).

Para isso FDR utiliza o dialeto CSP_M (*Machine-Readable* CSP), que é um padrão desenvolvido independentemente – apesar de ter sido primeiramente utilizada por FDR.

2.1.3 PAT e CSP#

PAT, ferramenta utilizada na elaboração deste trabalho, é um *framework* para criação, simulação e verificação de sistemas concorrentes, probabilísticos e de tempo real. Interessa para este trabalho apenas o primeiro deles, pois utiliza a linguagem CSP#, que mescla parte do formalismo de CSP apresentado por Hoare (1985) com facilidades de uma linguagem imperativa, como atribuição e compartilhamento de memória.

Alguns dos diferenciais do CSP# são: variáveis compartilhadas; comunicação assíncrona através de canais de tamanho definível; execução atômica de processos; suporte a *arrays* e outros tipos não-primitivos como listas e pilhas; suporte à programação imperativa interna aos eventos; possibilidade de incluir bibliotecas compiladas em C# com definições de tipos e métodos (SUN; LIU; DONG, 2011). Apesar de aparentemente exclusivas, parte dessas funcionalidades é passível de ser desenvolvida no CSP_M tradicional, mas pode tornar o modelo não escalável ou de pior legibilidade (CARVALHO et al., 2011).

Diferentemente da especificação do próprio CSP, CSP# não assume um ambiente, o que acaba influenciando na verificação de modelos. Por exempo, o Código 2.1 mostra um caso em que o processo P espera um valor de entrada, mas como não há nenhum outro processo que escreva neste mesmo canal, P não é livre de *deadlock*. Em CSP (e em CSP_M) isso não é verdade por que o ambiente estaria disposto a escrever no canal *entrada*.

Código 2.1: Exemplo mundo fechado de CSP#

```
1 channel entrada 0; channel saída 0;
2 P() = entrada?valor -> saída!valor -> P();
```

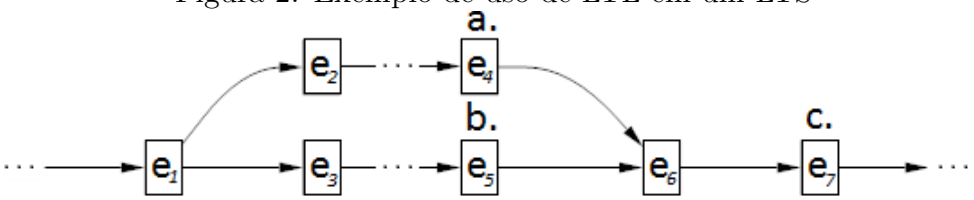
2.1.4 Lógica Temporal

PAT permite fazer asserções formuladas em lógica temporal linear (LTL) acerca da especificação CSP#. Na lógica classica, a avaliação de fórmulas se dá em um estado fixo de mundo, enquanto que na temporal, em um conjunto de estados (FISHER, 2011). Para descrever essa navegação pelo tempo no modelo, a lógica temporal extende os operadores clássicos. Por exemplo, a Equação 2.12 diz que $x \vee y$ é satisfeito no estado atual, mas que $x \wedge y$ é satisfeito no próximo.

$$(x \vee y) \bigcirc (x \wedge y) \tag{2.12}$$

Isso seria válido, por exemplo, em um sistema que podesse ser representado por um LTS, como mostra a Figura 2:

Figura 2: Exemplo de uso de LTL em um LTS



(a)x é satisfeito; (b) y é satisfeito; (c) $x \wedge y$ é satisfeito.

Tabela 1: Operadores LTL
[Fonte: Adaptado de SUN, LIU e DONG (2011)]

Textual	Simbólico	Explicação
$X\phi$	$\bigcirc\phi$	ϕ deve ser satisfeito no próximo estado
$G\phi$ ou $[]\phi$	$\Box\phi$	ϕ deve ser satisfeito em todos os estados subsequentes
$F\phi$ ou $<>\phi$	$\Diamond\phi$	ϕ deve ser satisfeito em algum estado subsequente
ψUϕ	$\psi u\phi$	ψ é satisfeito até que ϕ seja satisfeito
ψRϕ	$\psi R\phi$	ϕ é satisfeito até o primeiro estado em que ψ é satisfeito

São aceitos como entradas para verificações LTL em PAT: eventos, proposições pré-definidas e expressões com o conjunto extendido de operadores (SUN; LIU; DONG, 2011).

2.1.5 Ambiente de execução

Compiladores de linguagens de programação que apresentam um nível de abstração mais alto que o código de máquina nativo, como C, devem trabalhar em conjunto com o sistema operacional para criar um ambiente de execução apropriado.

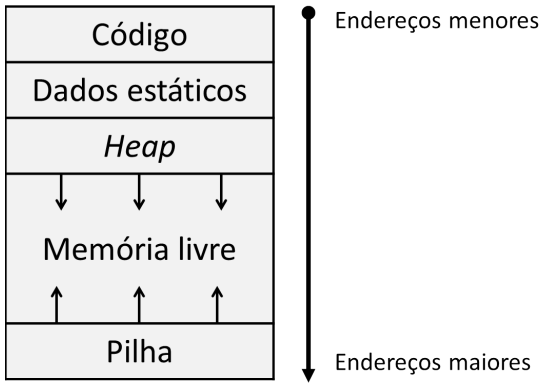
Tal ambiente é responsável, por exemplo, pelo posicionamento e esquema de alocação de variáveis descritas no código, pela passagem de parâmetros, pelo mecanismo de acesso

às variáveis, pela ligação entre os procedimentos, pelos dispositivos de entrada e saída, enfim, convenções utilizadas na execução (AHO et al., 2006).

2.1.5.1 Armazenamento

A Figura 3 mostra uma subdivisão da memória em diferentes áreas feita por um compilador de uma "linguagem de *von Neumann*". Assim conhecida por trabalharam em modelos computacionais baseados na arquitetura de *von Neumann*; cuja principal característica é de manter em memória tanto o programa, quanto os dados durante a execução.

Figura 3: Organização da memória
[Fonte: Adaptado de Aho et al. (2006)]



É importante notar como a parte livre da memória é compartilhada pela *heap* e pela pilha. Essas áreas crescem em direções opstas enquanto o programa está sendo executado.

Normalmente, o armazenamento nessas áreas está relacionado a diferentes finalidades: na pilha ficam variáveis de escopo local e dados necessários à ligação entre procedimentos; na *heap*, as variáveis que ocupam espaço sob demanda e podem durar além da vida de um escopo (AHO et al., 2006).

2.1.5.2 Pilha de execução

A pilha de execução é responsável, principalmente, por viabilizar a criação de *stack frames*, quadros de pilha, que representam os diferentes escopos de um programa, com o auxílio de dois registradores: o EBP (*Extended Base Pointer*), que aponta para o endereço da memória onde se encontra a base atual da pilha; e o ESP (*Extended Stack Pointer*), que aponta para o seu topo. O prefixo "*Extended*" dado aos registradores no *assembler* x86 denota que estes possuem 32 ao invés de 16 bits. À medida que os dados são empilhados,

o ESP é decrementado e passa a apontar para um endereço de memória de numeração menor.

No decorrer de uma execução, são empilhadas tanto variáveis locais como parâmetros para outros procedimentos e conteúdo de registradores, por exemplo, o EIP (*Extended Instruction Pointer*), também conhecido por PC (*Program Counter*). Os dois últimos casos, normalmente, são decorrência da chamada e retorno de procedimentos, pois o conteúdo dos mesmos é alterado com a mudança de escopo e deve ser restaurado ao estado anterior.

Não há uma divisão exata de como serão divididas as tarefas entre o procedimento chamador e o procedimento chamado. Pode haver variação até mesmo para diferentes implementações de compiladores de uma mesma linguagem (AHO et al., 2006).

As convenções utilizadas pelo GCC (*GNU Compiler Collection*) na plataforma Windows x64, em que este trabalho foi elaborado, são as seguintes:

- Os registradores EBX, ESI, EDI, EBP, DS, ES e SS não podem ter seu valor alterado dentro de uma chamada. Isso significa que, se usados, seus valores devem ser restaurados antes de retornar;
- Inteiros com tamanho de até 32 bits e ponteiros são retornados através do registrador EAX (*Extended Accumulator register*);
- Valores de ponto flutuante são retornados no registrador ST0 – registradores x87 (subconjunto de instruções relacionadas a operações com ponto flutuante que originalmente extendiam as x86) que vão de ST0 a ST8 e possuem 80 bits de armazenamento;
- Valores do tipo *long long int* são retornados nos dois registradores EDX (*Extended Data register*) e EAX, respectivamente contendo suas partes mais e menos significativas;
- O retorno de estruturas por valor pode causar erro de execução se houver mais de uma definição desse tipo;
- Os parâmetros são passados da direita para a esquerda;
- O endereço de retorno, passado pela pilha, fica na posição apontada por ESP.

Além dessas, a linguagem C permite que na declaração de uma função seja explicitado qual dentre suas possíveis convenções de chamada, referentes à passagem de parâmetros, será usada. Ver Tabela 2.

Tabela 2: Convenções de chamada em C
[Fonte: Adaptado de MSDN[1]]

Tipo	Descrição
stdcall	Função **chamada** é responsável por limpar argumentos da pilha. Gera assembly com nome decorado com a quantidade de *bytes* de seus argumentos.
cdecl	Esta é a convenção padrão para compilação de códigos-fonte escritos em C. Função **chamadora** fica responsável por limpar argumentos passados.
fastcall	Os argumentos são passados através dos registradores ECX (*Extended Count register*) e EDX quando possível. Gera assembly com nome decorado com a quantidade de bytes em argumentos. Função **chamada** fica responsável por limpar seus argumentos.

2.1.5.3 Gerenciamento da *Heap*

A *heap* é responsável pelos dados gravados na memória por tempo indefinido ou até que sejam explicitamente desalocados. O gerenciador da *heap* é o subsistema responsável pelo uso dessa parte da memória. Seu trabalho é o de guardar informações sobre partes ainda livres da memória, distribuindo-as e retomando-as, através de duas funções básicas de alocar e desalocar. Sempre com a preocupação de minimizar o espaço da *heap* necessário a um programa e de maximizar a eficiência do mesmo. Porém, nem todos os pedidos de partes da memória são do mesmo tamanho e suas devidas liberações acabam gerando espaços vazios que podem não ser mais preenchidos posteriormente.

2.1.6 Assembly

O desenvolvimento de software é normalmente feito em linguagens de programação que fornecem certo nível de abstração ao programador; no sentido de que não é preciso conhecer a fundo a linguagem de máquina para poder comandá-la. Isso favorece principalmente a legibilidade, a portabilidade e a manutenabilidade dos programas.

Porém, um computador é, "a grosso modo", um circuito digital; o que significa que os dados manipulados por ele são representados por sinais eletrônicos que assumem apenas dois estados (HENNESSY; PATTERSON; LARUS, 2000). Portanto, o vocabulário de uma máquina está restrito ao seu conjunto de instruções.

[1]MSDN *Calling Convention Topics*: http://msdn.microsoft.com/en-us/library/aa278874(v=vs.60)

Uma instrução é uma sequência ordenada de bits que determina o fluxo dos dados de entrada fornecidos. Há várias categorias de instruções, dentre elas as aritméticas, as de transferência de dados e as de desvio condicional e incondicional. Normalmente essas instruções estão associadas a um mnemônico que substitui a necessidade de memorizar a sequência de bits, além de servir como um pequeno nível de abstração.

2.1.6.1 Instruções

As instruções na Tabela 3 são um subconjunto das instruções disponíveis para Assembly x86. Todas as instruções encontradas nos exemplos apresentados posteriormente são listadas.

2.1.6.2 Diretivas

Um arquivo Assembly contém ao menos três seções não necessariamente preenchidas: de texto, de variáveis estáticas inicializadas e não inicializadas dispostas nesta ordem. Contudo, tais seções podem aparecer de maneira alternada, através do uso explícito de suas respectivas diretivas *text*, *data* e *bss* (*Block Started by Symbol*).

A diretiva *comm*, que também aparece nos exemplos deste trabalho, declara e aloca memória não inicializada na seção *bss* para um símbolo (JURIC; REICHELT; KOFLER, 2003). Já a diretiva *globl* (ou *global*) torna um símbolo visível para todos os demais programas parciais sendo compilados, ou seja, outros objetos assembly poderão acessá-lo pelo seu nome.

Nó exemplo do Código 2.2 tanto a variável *a* quanto a *b* ficarão no segmento *bss*, mas, por já ter sido inicializada, *b* já será taxada como global. Enquanto isso, a variável estática *c* será posicionada diretamente em *data*.

Código 2.2: Variáveis estáticas em C

```
1 int *a;
2 int b = 0;
3 static int c = 1;
```

Tabela 3: Instruções Assembly

Mnemônico	**Operação**
mov	Move o conteúdo do primeiro operando para o segundo.
add	Adiciona o primeiro operando ao segundo. Guarda o resultado no segundo.
sub	Subtrai o primeiro operando do segundo. Guarda o resultado no segundo.
mul	Multiplica conteúdo do registrador A (por exemplo, EAX) pelo único operando. Guarda o resultado no operando.
imul	Mesmo que mul, mas considera sinal.
dec	Decrementa o conteúdo do único operando.
inc	Incrementa o conteúdo do único operando.
push	Empilha o único operando e atualiza o registrador ESP.
pop	Desempilha preenchendo o conteúdo do único operando.
jmp	Desvia a execução, altera o PC, para o endereço do *label*.
call	Executa um *push* do PC e um *jmp* com o argumento (*label*).
ret	Desempilha preenchendo o PC, retornando a execução para o ponto anterior. Há um parâmetro inteiro opcional que quando passado é somado ao ESP.
leave	Move EBP para ESP e desempilha o valor de EBP antigo.
lea	Carrega o valor apontado pelo primeiro operando no segundo.
cmp	Faz uma subtração entre os operandos e atualiza o registrador de *flags*.
je	Desvia para *label* quando operandos do *cmp* anterior são iguais.
jne	Desvia para *label* quando operandos do *cmp* anterior são diferentes.
jg	Desvia para *label* quando o primeiro operando do *cmp* anterior é maior.
jge	Desvia para *label* quando o primeiro operando do *cmp* anterior é maior ou igual.
jl	Desvia para *label* quando o primeiro operando do *cmp* anterior é menor.
jle	Desvia para *label* quando o primeiro operando do *cmp* anterior é menor ou igual.

2.1.7 Concorrência

2.1.7.1 Condições de corrida

Alguns sistemas computacionais possibilitam que múltiplas ações sejam realizadas de forma independente em cima de um mesmo recurso. Se ao menos uma dessas ações for de escrita em memória compartilhada, a integridade desses dados pode ser comprometida.

Condições de disputa são falhas dinâmicas, que acontecem em tempo de execução e são dificilmente detectados diretamente no código fonte (HENNESSY; PATTERSON; LARUS, 2000). Elas são decorrentes da natureza multitarefa dos sistemas e da maneira como são feitos os escalonamentos pelo sistema operacional.

É comum que uma linha de código fonte seja compilada tornando-se mais de uma linha de código de máquina – nível onde ocorrem as trocas de contexto entre tarefas. Portanto, não há garantias, por exemplo, de que o valor lido e copiado para um registrador em um momento não tenha sido alterado imediatamente depois por outra tarefa, gerando inconsistência no dado do registrador.

Um fator atenuante para esse tipo de falha é que o número de combinações de entrelaçamento de tarefas que a geram tende a ser relativamente limitado.

2.1.7.2 Regiões críticas

As seções, ou regiões, críticas são trechos de código em que tarefas acessam dados compartilhados, ou seja, que podem gerar condições de disputa. Uma vez identificada, há diversas maneiras de evitar esse tipo de problema restringindo o acesso à mesma para apenas uma tarefa por vez.

De maneira ideal, o acesso controlado a essas regiões deve contemplar: a exclusão mútua, apenas uma tarefa na região; a espera limitada, garantindo que uma tarefa não esperará indefinidamente; a independência de outras tarefas, pois apenas tarefas que desejam entrar na região crítica devem influenciar na decisão de quem tomará o controle; e a independência de fatores físicos, porque componentes de *hardware* não podem influenciar na decisão (MAZIERO, 2011).

2.1.7.3 Impasses

O não cumprimento de todas as condições para acesso seguro aos recursos pode levar o sistema, ou parte dele, a estados de impasse (MAZIERO, 2011). Esses estados são conhecidos, de modo generalizado, como *deadlocks*. Porém, há diferentes situações de impasse, todas elas impedindo o progresso da execução de um programa.

O *deadlock* é, mais especificamente, um caso no qual a tarefa fica bloqueada esperando a liberação ou a obtenção de algum recurso para continuar sua execução.

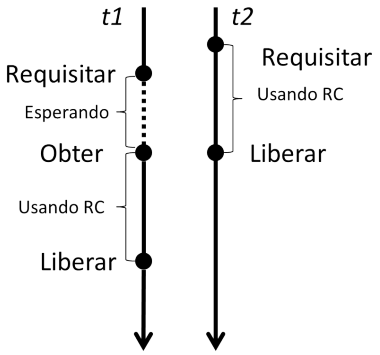
Já o *livelock*, apesar de parecido com o *deadlock*, apresenta uma característica importante. Ao invés de estar presa a um estado, a tarefa fica presa a um conjunto de estados que nunca progride, como em um laço infinito, tentando obter o recurso.

2.1.7.4 *Mutexes*

Uma das formas de controlar o acesso a seções críticas de um programa é através do uso de *mutexes*. Pois com eles é possível marcar uma região como ocupada ou livre. Assim, antes de adentrar uma região crítica, toda tarefa deve, primeiramente, pedir o travamento (*lock*) da mesma àquele *mutex* e esperar que ele lhe dê permissão para isso. Ao terminar de usá-la, o *mutex* deve ser então notificado para que se dê chance às outras tarefas (MAZIERO, 2011).

Na Figura 4 duas *threads* $t1$ e $t2$ fazem requisições para entrar na região crítica onde poderiam, por exemplo, incrementar ou decrementar o valor de uma variável compartilhada. Se tal região não fosse protegida, ambas poderiam ler o mesmo valor inicial, mas apenas a última delas a guardar o novo valor aparentaria ter ocorrido.

Figura 4: Funcionamento do *Mutex*



2.1.7.5 Manipulação de *Threads* em C no Windows

Threads (linhas de execução) são as menores unidades de processamento escalonáveis pelo sistema operacional. Elas são internas ao escopo do processo para o qual foram criadas e compartilham os recursos do mesmo, como memória e sequência de instruções. Desta forma, várias linhas de execução de um mesmo processo rodam de maneira paralela, podendo gerar condições de disputa.

A implementação de *threads* varia para cada sistema operacional, fazendo com que sua chamada também tenha particularidades dependentes do mesmo. Para manipulá-las no ambiente Windows é preciso incluir o cabeçalho "windows.h", que inclui, internamente, o "winbase.h", onde estão definidas as funções da Tabela 4.

Tabela 4: Funções utilizadas por este trabalho

Nome	Descrição
CreateThread	Cria uma *thread*.
SuspendThread	Suspende a execução de uma *thread*
ResumeThread	Retoma a execução de uma *thread*
ExitThread	Finaliza uma *thread*
CreateMutex	Cria um *mutex*
WaitForSingleObject	Espera a liberação de um objeto (e.g. *threads, mutexes*)
WaitForMultipleObjects	Espera a liberação de um ou mais objetos

Para criar *threads* utiliza-se a função *CreateThread*, que possui seis argumentos, sendo dois deles opcionais e um de saída, como mostra o Código 2.3. A Tabela 5 mostra algumas das definições da API (*Application Programming Interface*) do Windows que aparecem nessa assinatura ou que são usadas nos exemplos do trabalho posteriormente. As demais funções utilizadas neste trabalho podem ser encontradas na *Microsoft Development Network* (MSDN), mais especificamente na área de serviços de sistema[2].

Código 2.3: Assinatura da função *CreateThread*

```
1 HANDLE WINAPI CreateThread(
2   __in_opt LPSECURITY_ATTRIBUTES lpThreadAttributes,
3   __in SIZE_T dwStackSize,
4   __in LPTHREAD_START_ROUTINE lpStartAddress,
5   __in_opt LPVOID lpParameter,
6   __in DWORD dwCreationFlags,
7   __out_opt LPDWORD lpThreadId);
```

[2]MSDN *System Services*:
http://msdn.microsoft.com/en-us/library/windows/desktop/ee663297(v=vs.85).aspx

Tabela 5: Tipos definidos na API de C do Windows
[Fonte: Adaptado de MSDN[3]]

Tipo	Descrição
HANDLE	Manipulador para um objeto. É um tipo definido por PVOID.
PVOID/LPVIOD	Ponteiro para qualquer tipo (ponteiro para *void*).
WINAPI	Convenção de chamada. Será substituído por *stdcall* pelo pré-processador.
DWORD	Inteiro sem sinal de 32 bits.
LPDWORD	Ponteiro para um DWORD.
SIZE_T	Número máximo de *bytes* que um ponteiro pode apontar.
LPSECURITY_ATTRIBUTES	Ponteiro para estrutura SECURITY_ATTRIBUTES.
SECURITY_ATTRIBUTES	Estrutura que define configurações de segurança para objetos criados por algumas funções como o *CreateThread*.
LPTHREAD_START_ROUTINE	Ponteiro para função de *callback*.
INFINITE	DWORD máximo (4294967295).
BOOL/BOOLEAN	*Byte*/inteiro sem sinal. Devem ser TRUE ou FALSE.
TRUE	Equivalente ao valor 1.
FALSE	Equivalente ao valor 0.
NULL	Equivalente a \0.
_in[_opt], _out[_opt]	Pré-fixos de parâmetros que apenas servem de informação. Serão excluídos pelo pré-processador.

2.2 Trabalhos Relacionados

A idéia de utilizar o código Assembly para fazer verificação de *software* já vem sendo explorada por outros autores. Essa verificação é bastante desafiadora. Os motivos incluem a não-estruturação do código, a não utilização de tipos de variáveis, o uso de registradores para guardar tanto valores quanto endereços de memória e o fato de que algumas instruções podem alterar o estado do processador (MAUS; MOSKAL; SCHULTE, 2008). Esta seção mostra três soluções que, de certa forma, se assemelham com a idéia deste trabalho.

2.2.1 Vx86

Vx86 (MAUS; MOSKAL; SCHULTE, 2008) é uma ferramenta de análise estática automática que verifica a corretude de código Assembly x86 da Intel simulando-o em C e passando por um provador de teoremas. Basicamente o Assembly decorado é traduzido para um C também decorado que é entendido e compilado pela ferramenta na qual sua base foi construída, o VCC (*Verifying C Compiler*).

O VCC (DAHLWEID et al., 2009) é capaz de verificar a corretude parcial de código C decorado. Isso significa que a verificação pode não terminar, é uma tarefa indecidível, mas

[3]MSDN Windows *Data Types*: http://msdn.microsoft.com/en-us/library/aa383751

se alguma resposta for retornada então a condição de verificação requisitada foi provada ou refutada.

Na verdade o VCC também traduz seu código anotado para outra linguagem, a BoogiePL, que é uma linguagem simples para propósito de verificação de programas orientados a objetos. Essa, por sua vez, traduz as funções em conjuntos de condições de verificação em lógica de primeira ordem e usa o solucionador SMT (*Satisfiability Modulo Theories*) Z3 para provar a validade dessas fórmulas.

As propriedades de boa-formação passíveis de verificação são: segurança de memória, checa se nenhuma função tenta acessar endereços que não são conhecidamente válidos; segurança de aritmética, verifica ausência de *overflows*; segurança de chamada, testa se a pilha sempre é limpa depois de cada chamada e se os registradores são devidamente salvos antes dela; segurança de interrupção, checa se a pilha é limpa depois de processar a interrupção por completo.

Apesar de basear-se no VCC, capaz de fazer verificações em cima de códigos concorrentes, o Vx86 só faz verificação de código Assembly garantidamente sequencial. Portanto, apesar da similaridade com este trabalho em tentar formalizar a verificação de código, Vx86 não considera o paralelismo.

2.2.2 LLVM2CSP

LLVM (*Low Level Virtual Machine*) é uma infraestrutura compiladora desenvolvida para fornecer informações de alto nível para as transformações feitas por compiladores em tempo de compilação, ligação, execução e inatividade entre compilações (LATTNER; ADVE, 2004). Por ser livre de linguagem, vem sendo usada por uma variedade cada vez maior de compiladores.

O LLVM define uma representação de código abstrata de baixo nível com conjunto de instruções parecida com a de um RISC (*Reduced Instruction Set Computer*), mas que provê informações adicionais como tipos primitivos de linguagens de alto nível, aritmética de endereços tipada, manipulação de exceções.

Tomando como base a representação intermediária de LLVM para códigos concorrentes escritos originalmente em C ou C++, o LLVM2CSP (LLVM *to* CSP) desenvolvido por Kleine et al. (2011) é uma ferramenta para LLVM que extrai modelos CSP tanto para ferramenta ProBE (*Process Behaviour Explorer*) quanto FDR.

As instruções são traduzidas em processos sequenciais que terminam. A memória é dividida em privada e compartilhada e ambas são modeladas como processos separados. É definido também um processo que simula o escalonamento de tarefas, que, em conjunto com os processos que descrevem o funcionamento da memória, forma a semântica operacional do LLVM em CSP.

O modelo gerado é dividido em três partes: uma parte específica da aplicação, descreve o comportamento de *threads*; outra de domínio específico, que encapsula conceitos de baixo nível como escalonamento; e uma última específica de plataforma, que contém o modelo do *hardware*. Algumas dessas partes possuem implementação pré-definida, enquanto outras tem os parâmetros ou são inteiramente geradas pelo LLVM2CSP.

As asserções sobre o modelo podem ser criadas através da própria decoração do código C e aparecerão no arquivo CSP_M final. Com o modelo em mãos, é possível fazer, por exemplo, verificações de refinamento com FDR ou de fórmulas LTL com o ProBE.

2.2.3 Mapeamento de Bytecode Java para CSP#

Lima (2011) propôs um mapeamento de *bytecode* Java para CSP# no qual o presente trabalho foi baseado. Sua justificativa para o uso não do código fonte em si, mas sim do *bytecode* gerado pela JVM é de que escalonamento entre as *threads* em Java se dá neste nível.

Neste trabalho foram extraídas regras gerais, regras de nomenclatura e regras de mapeamento de instruções a partir de exeperiências na tradução manual de códigos usados como exemplos. Um deles continha um problema de condição de corrida utilizado como demonstração, o qual conseguiu ser detectado com sucesso através da ferramenta de análise fornecida pelo *framework* PAT em cima do modelo gerado.

Porém, uma linha de *bytecode* Java, pode acabar sendo traduzida em mais de uma linha de código de máquina na hora da execução. Surgiu, portanto, a preocupação quanto ao que aconteceria à validade dessas verificações se parte do *bytecode* analisado fosse compilado pelo JIT (*Just In Time Compiler*) ao invés de inteiramente interpretado durante a execução.

Quando um método codificado em Java é compilado, a JVM alimenta o JIT com o *bytecode* gerado. O JIT, então, procura entender sua semântica e sintaxe para decidir se aquela parte do código será otimizada e compilada para código nativo da plataforma em que a JVM está sendo executada. Com isso, a máquina virutal Java não precisará

passar por todas as fases novamente para executar aquele trecho de código. Mesmo que a execução do mesmo esteja vinculado de alguma maneira pela JVM com o devido *bytecode* que o gerou, a manipulação de *threads* é uma funcionalidade normalmente provida pelo sistema operacional, é ele o responsável pelas trocas de contexto que acontecem no(s) processador(es).

3 Método de Pesquisa

Este capítulo descreve o método de pesquisa utilizado e como foram elaboradas e executadas as diferentes etapas da confecção deste trabalho a fim de alcançar os objetivos descritos no Capítulo 1.

3.1 Qualificação do Método de Pesquisa

Como pode ser observado no Capítulo 2, a fundamentação teórica necessária para o entendimento deste trabalho é extensa. Trata-se de uma pesquisa bibliográfica, elaborada a partir de conceitos já publicados em livros, artigos, manuais e outros.

3.2 Etapas do Método de Pesquisa

Estudo de CSP: Inicialmente foram estudados desde os conceitos mais básicos de CSP, como eventos e processos, até os operadores de paralelismo que seriam utilizados neste trabalho, e dois diferentes dialetos: CSP# e CSP$_M$. Após isso, era necessário definir para qual desses seria feito o mapeamento, já que suas implementações diferem bastante tanto em sintaxe quanto no próprio funcionamento. Optou-se, então, por CSP# principalmente pela maior simplicidade nos modelos com o uso de operações sobre variáveis compartilhadas. Outro fator que contribuiu para essa escolha foi a conveniência de as ferramentas de verificação e desenvolvimento serem partes de um *framework* executado na mesma plataforma para qual o *Assembly* seria gerado – Windows.

Estudo sobre compiladores: Nesta etapa foram estudados aspectos do ambiente de execução que estão relacionados às decisões dos compiladores durante a geração do código *Assembly* a partir de arquivos fonte em linguagens de mais alto nível. Em especial as convenções de chamada, a alocação e crescimento de memória (*heap* e pilha), a manipulação dos registradores EBP e ESP, entre outros.

Além disso, apesar de o mapeamento ser a partir do *Assembly*, os exemplos não são escritos diretamente neste nível, pois é bastante suscetível a erros de programação e difícil de manter e de corrigir. Portanto, foi estudado o funcionamento básico do GCC para a geração desses códigos a partir de fontes desenvolvidos em C.

Estudo de *Assembly*: Aqui foram estudados as diversas instruções responsáveis pela real execução do programa. Seu estudo é importante pois cada uma tem um ou mais comportamentos que devem ser também diferenciados durante o mapeamento. Muitas vezes esse comportamento altera implicitamente o estado do processador e sem conhecê-lo não é possível criar um modelo que seja fiel à realidade.

Estudo sobre concorrência: Como cada sistema operacional fornece seus próprios meios de manipulação de tarefas concorrentes, foi preciso estudar a API específica da plataforma Windows para desenvolver os exemplos usados no trabalho. Isso inclui principalmente a criação, suspensão e retomada de *threads* e a criação e uso de *mutexes*.

Estudo de Trabalhos Relacionados: Foram estudadas três soluções propostas por outros autores cujos objetivos eram similares ao deste trabalho. Além de servir como experiência adicional acerca dos problemas a serem enfrentados, o estudo permitiu traçar algumas vantagens e desvantagens de cada solução, bem como diferenças das mesmas em relação à apresentada aqui.

Definição dos exemplos de programa concorrente: Escolheu-se arbitrariamente dois problemas clássicos de concorrência como exemplos a serem mapeados e analisados neste trabalho: produtores/consumidores e o jantar dos filófos. Ambos foram desenvolvidos em C e apresentam problemas que, quando reproduzidos, os levam a nunca terminar.

Extração de regras de mapeamento: As regras foram extraídas tanto da compilação de códigos mais simples – até mesmo não-concorrentes – quanto de traduções diretas do *Assembly* dos próprios exemplos. Assim, a partir dos modelos obtidos, tentou-se estabelecer padrões de mapeamento que funcionassem de forma a abrangir o conjunto base desses códigos em sua completude.

Análise dos resultados: Após a aplicação das regras de mapeamento, os modelos CSP# são submetidos à ferramenta de análise fornecida pelo PAT, onde são testadas as propriedades: *deadlockfree* e *non-terminating*. A primeira verifica se há ou não situações que levam a impasses, que impedem a evolução do estado do programa. A segunda, se o programa nunca termina sua execução; havendo ao menos um caso de terminação, essa propriedade é falsa, incluindo um caso de *deadlock*. Foi utilizada também a verificação por asserções LTL no exemplo dos filósofos.

4 Resultados

Este capítulo mostra a representação de uma máquina x86 em PAT e o conjunto de regras definido por este trabalho para o mapeamento de código *Assembly* x86 em especificações CSP#. Em seguida, são mostrados os resultados das análises feitas sobre as especificações, estas obtidas após aplicação das regras nos seguintes exemplos: Produtores e Consumidores e Jantar dos Filósofos.

4.1 x86 em PAT

Esta seção define a representação de uma máquina x86 no ambiente de PAT para permitir a tradução e simulação/verificação de código Assembly.

4.1.1 Modelo de Memória

A memória do sistema será representada por uma matriz $(n+1)$ x m onde n é o número máximo pré-definido de *threads* permitidos, incluindo a *main*, e m é o tamanho um máximo pré-definido daquele fragmento da memória. As n primeiras dimensões são dedicadas às pilhas de execução das *threads* de identificador correspondente. n é acrescido de um pois a *heap* também será mapeada nesta matriz sempre na última dimensão, compartilhada por todas as *threads*. Portanto, a estrutura adotada não admite fragmentos de tamanhos diferentes. Deve-se, logo, assumir que o fragmento será do tamanho do maior fragmento existente no código *Assembly*.

O Código 4.1 mostra também como é inicializada essa matriz, que representa a memória do computador no modelo em CSP#. As macros definidas no início do código são substituídas pelo pré-processador do PAT, assim como em uma compilação de arquivos C.

Código 4.1: Memória

```
1 #define MEM_SIZE 30;
2 #define TOTALTHREADS 3;
3 #define HEAP TOTALTHREADS;
4 var memory[TOTALTHREADS + 1][MEM_SIZE];
```

4.1.2 Modelo dos Registradores

Para todos os registradores utilizados pelo *Assembly* (EAX, EBX, ECX, etc) é criado um vetor com seu nome de forma que são acessados através do índice, identificador, da *thread*.

O Código 4.2 mostra como são declarados e inicializados os registradores no modelo CSP#. Normalmente os registradores são inicialmente preenchidos com 0 (zero). Excepcionalmente, os registradores ESP e EBP são inicializados apontando para o último índice ($MEM\_SIZE - 1$). A notação que aparece nas linhas 4 e 5 inicializam o vetor com o valor MEM_LAST_INDEX repetindo-o $TOTALTHREADS$ vezes.

Código 4.2: Registradores

```
1 #define TOTALTHREADS 3;
2 #define MEM_LAST_INDEX 29;
3 var eax[TOTALTHREADS];
4 var ebp = [MEM_LAST_INDEX(TOTALTHREADS)];
5 var esp = [MEM_LAST_INDEX(TOTALTHREADS)];
```

Já o Código 4.3 mostra os exemplos de uma instrução "*mov* $\%esp, \%ebp$", dentro da main, e outra "*sub* $\$2, \%eax$", em uma função de *thread*, usando os registradores.

Código 4.3: Uso dos registradores

```
1 #define MAIN 0;
2 var eax[TOTALTHREADS];
3 _main() = (...) -> _main_mov { ebp[MAIN] = esp[MAIN] } -> (...);
4 _function(id) = (...) -> _function_sub.id {eax[id] = eax[id] - 2} -> (...);
```

Dessa forma, o escalonamento das *threads*, que salva e restaura estados dos registradores, pode ser simulado dentro do modelo CSP. A restrição é nunca utilizar identificadores diferentes do que está sendo atualmente executado, pois isso geraria incoerência dos dados.

4.1.3 Modelo das Variáveis Globais

Variáveis de escopo global identificadas dentro do código *Assembly* são mapeadas como variáveis de ambiente no CSP# com o mesmo nome e apontam para uma posição

específica da memória. A identificação é feita procurando pelas diretivas *comm, globl e bss* descritas anteriormente. O *assembler* faz a distinção dos segmentos *data*, *bss* e *heap* durante a criação dessas variáveis. Porém, como isso não alteraria o comportamento final das mesmas, decidiu-se que o mapeamento as colocasse sempre na região equivalente à *heap*.

Código 4.4: Identificação de variáveis globais

```
1 .globl _x
2         .bss
3         .align 4
4 _x:
5         .space 4
6 (...)
7 .comm   _y, 16   # 8
```

No Código 4.4 a variável *_x* de 4 *bytes* (especificada pela diretiva *space* na linha 5) e a variável *_y* de 8 *bytes* (valor após o # na linha 7) são definidas. Apesar da identificação ser diferente, o resultado do mapeamento é o mesmo. Para ambas uma posição inicial na *heap* é reservada. Como *_y* tem tamanho maior que 4 *bytes*, e no CSP# não há tipos de 64 bits, há a necessidade de ocupação de dois espaços, portanto, o índice conseguinte é pulado para futuras alocações, deixando-o livre também para o acesso deslocado através dessa variável.

Identificadas as duas variáveis globais, antes de iniciar a execução própria do corpo de *main*, o processo principal se comporta como o *DefineGlobalVars*, que possui eventos de *malloc* consecutivos para cada uma das variáveis globais. O CSP# resultante é o encontrado no Código 4.5. A limpeza no registrador EAX garante que essa manobra não afete estados posteriores da execução.

Código 4.5: Definição das variáveis globais

```
1 var _x;
2 var _y;
3 DefineGlobalVars() =
4         Malloc(MAIN, 12)
5         ; _main_movl { _x = eax[MAIN] } ->
6         Malloc(MAIN, 12)
7         ; _main_movl { _y = eax[MAIN] } ->
8         _main_movl { eax[MAIN] = 0 } -> Skip;
```

4.1.4 Código não mapeado

O código gerado pelo GCC na plataforma Windows acrescenta algumas instruções de alinhamento e configuração da biblioteca da linguagem que são desconsideradas para o escopo deste trabalho. O exemplo de programa em C mostrado no Código 4.6 foi compilado através do comando "gcc -S" e o *Assembly* obtido consta no Código 4.7.

Código 4.6: Exemplo de programa simples

```
1  #include <stdlib.h>
2  int main (void){
3          int a, b;
4          a = 2;
5          b = 3;
6          return a + b;}
```

Código 4.7: Resultado da compilação

```
1          .file   "ExemploC.c"
2          .def    ___main;        .scl    2;      .type   32;     .endef
3          .text
4  .globl _main
5          .def    _main;  .scl    2;      .type   32;     .endef
6  _main:
7          push    %ebp
8          mov     %esp, %ebp
9          sub     $24, %esp
10         and     $-16, %esp
11         mov     $0, %eax
12         add     $15, %eax
13         add     $15, %eax
14         shr     $4, %eax
15         sal     $4, %eax
16         mov     %eax, -12(%ebp)
17         mov     -12(%ebp), %eax
18         call    __alloca
19         call    ___main
20         mov     $2, -4(%ebp)
21         mov     $3, -8(%ebp)
22         mov     -8(%ebp), %eax
23         add     -4(%ebp), %eax
24         leave
25         ret
```

O segmento de código não mapeado é o intervalo fechado das linhas $[10, 19]$ – começando no *and* e terminando no *call* ____*main*. É importante observar que o estado dos registradores EBP e ESP, e da memória *heap* não são alterados dentro deste bloco; e que, apesar de ser modificado dentro, a próxima operação em EAX fora do bloco é de escrita.

Outro caso de não mapeamento do código acontece nas linham que sucedem chamadas externas às funções da API do Windows, pois estas usam a convenção de chamada *stdcall*. Logo, internamente, presume-se que tais funções retornam usando *ret N*, onde *N* é o espaço ocupado pelos parâmetros em *bytes*, removendo-os da pilha; portanto, o *sub* correspondente àquele retorno, como no Código 4.8, é excluído do mapeamento.

Código 4.8: Exemplo de chamada usando convenção *stdcall*

```
1 (...)
2 call    _CreateThread@24
3 sub     24, esp
4 (...)
```

4.2 Regras de Mapeamento de Chamadas Externas

Todas as chamadas a funções que não são definidas dentro do mesmo arquivo *Assembly* se tornarão processos separados. Para os dois exemplos estudados neste tabalho, as únicas funções externas usadas são: *malloc, CreateThread, ResumeThread, SuspendThread, CreateMutex, WaitForSingleObject* e *WaitForMultipleObjects*. São definidas as seguintes Regras de Chamadas Externas (RCE) para a simulação da execução de código x86 no ambiente de PAT.

4.2.1 RCE 1: Alocação de Memória

A alocação de memória *heap* suporta somente chamadas à função *malloc*. Não há suporte à liberação de espaço alocado para posterior reutilização, nem ao rearranjo para compactação do espaço utilizado, pois não há gerenciamento de estado para posições da memória. Alocações consecutivas receberão endereços crescentes, começando de 0 (zero) dentro da dimensão da memória compartilhada dedicada à *heap*. O processo que simula esse comportamento é mostrado no Código 4.9.

Código 4.9: Alocação das variáveis

```
1 var current_heap = 0;
2 Malloc(id, size) =
3         call_malloc.id {
4                 eax[id] = current_heap;
5                 current_heap = current_heap + size/4;
6         } -> Skip;
```

4.2.2 RCE 2: Criação de *Threads*

Assim como a alocação de memória, a chamada ao *CreateThread* é mapeada como um processo desenvolvido de modo a simular de maneira razoável o seu funcionamento. Ele recebe dois parâmetros: o nome do procedimento e o identificador da *thread* que o chamou. O primeiro parâmetro é, na verdade, uma macro que relaciona unicamente o procedimento (*label* no código) a um número inteiro – sempre colocando o sufixo *_Proc* em seu nome. O segundo é usado para modificar o conteúdo do registrador EAX de quem cria.

Para simular a espera e retomada das *threads*, seus estados atuais (*threadState*) são guardados em um vetor e são criados canais de comunicação (*ResumeThread_channel*) para cada uma. O estado da *thread* usado por ambos os canais pode ser 0 (parada), 1 (executando) ou 2 (terminada). O estado da *main* é iniciado como 1, enquanto as demais *threads* tem valor inicializado em 0. Mais detalhes são dados na descrição da RCE 3 e da RCE 4.

O retorno real dessa função é um *HANDLE* para a *thread*, mas no CSP é retornado apenas o inteiro identificador da mesma. O fato de ser atômico garante que os dois eventos sejam comunicados fazendo com que a *thread* esteja pronta para iniciar sua execução e o criador agora esteja pronto para executar sua próxima instrução simultaneamente.

Os comentários que aparecem no Código 4.10 mostram que o quarto parâmetro (12 *bytes* acima de ESP) do *CreateThread* é empilhado para a nova *thread*. Em seguida, o estado da *thread* passa de parado para executando. A partir de então, espera-se o início efetivo da sua execução com a sincronização do evento *start*.

Código 4.10: Mapeamento do *CreateThread*

```
1  var threadState = [1, 0(MAX_THREADS)];
2  var current_id = 0;
3  _CreateThread (proc, creatorId) =
4   atomic {
5    call_CreateThread {
6     current_id++;
7     esp[current_id] = esp[current_id] - 1; //sub 4, %esp
8     // Passagem 4º do parâmetro do CreateThread para função de CallBack
9     memory[current_id][esp[current_id]] = memory[creatorId][esp[creatorId] + 3];
10    esp[current_id]--; memory[current_id][esp[current_id]] = pc_dummy; // push %PC
11    eax[creatorId] = current_id; // Coloca id no retorno
12   } -> setThreadState.id { threadState[id] = 1 } ->
13                 start.proc.(eax[creatorId]) -> Skip};
```

4.2.3 RCE 3: Suspensão e retomada de execução de *Threads*

A presença de chamadas às funções *ResumeThread* e *SuspendThread* acarreta na adição dos processos pré-definidos descritos no Código 4.11, além das variáveis e canais de controle descritos na RCE 2. As operações são atômicas para garantir que não haja interferência externa à execução destes processos. A linha onde ocorre a chamada a uma dessas funções é mapeada como seu respectivo processo com o identificador da *thread* como parâmetro.

Código 4.11: Suspensão e retomada

```
1 _ResumeThread(id) =
2         atomic {
3                 if(threadState[id] == 1) { Skip }
4                 else {
5                         setThreadState.id { threadState[id] = 1 } ->
6                         ResumeThread_channel[id]!1 -> Skip} };
7 _SuspendThread(id) =
8         atomic{ setThreadState.id { threadState[id] = 0 } ->
9                 ResumeThread_channel[id]?1 -> Skip };
```

4.2.4 RCE 4: Esperas

As esperas são, mais uma vez, referentes a chamadas de funções presentes na API do Windows que são mapeadas de maneira a simular o comportamento real em processos pré-definidos. São elas o *WaitForSingleObject* e o *WaitForMultipleObjects*.

A espera por objetos é usada tanto com *mutexes* quanto com *threads* – ambos do tipo *HANDLE* –, porém os mapeamentos dos dois casos diferem. Para distingui-los deve-se atentar ao primeiro parâmetro (diretamente apontado por ESP durante o empilhamento). Se tal parâmetro contém o mesmo endereço apontado pelo resultado de um *CreateMutex* anterior e não houve nenhuma modificação nele, então o mapeamento utilizado será o exclusivo às esperas por *mutexes*. Nesse caso é comum o aparecimento também de um *ReleaseMutex* referenciando o mesmo endereço. Essas chamadas são mapeadas apenas em substituí-las pelos eventos de escrita nos canais $LockMutex$ e $ReleaseMutex$ definidos no Código 4.15.

No exemplo do Código 4.12, foi criada uma função genérica *_exemplo* que apenas faz a chamada ao *WaitForSingleObject* e ao *ReleaseMutex*. Pode-se ver que *_mutex* é colocado em EAX, que, em seguida, é empilhado exatamente em ESP – espaço reservado para o argumento *HANDLE* da função de espera. A última linha, mostra que *_mutex*

é, na verdade, uma variável global. Observando a da definição de *_main*, vê-se que o *HANDLE* apontado por esse endereço é o resultado de um *CreateMutex*.

Código 4.12: Exemplo de uso *WaitForSingleObject* em *Mutexes*

```
1  _exemplo:
2          push    %ebp
3          mov     %esp, %ebp
4          sub     $8, %esp
5          mov     $-1, 4(%esp)
6          mov     _mutex, %eax
7          mov     %eax, (%esp)
8          call    _WaitForSingleObject@8
9          sub     $8, %esp
10         (...)   #Região protegida pelo _mutex#
11         mov     _mutex, %eax
12         mov     %eax, (%esp)
13         call    _ReleaseMutex@4
14         sub     $4, %esp
15         leave
16         ret
17         _main:
18         (...)
19         mov     $0, 8(%esp)
20         mov     $0, 4(%esp)
21         mov     $0, (%esp)
22         call    _CreateMutexA@12
23         sub     $12, %esp
24         mov     %eax, _mutex
25         (...)
26         .comm   _mutex, 16      # 4
```

Já no Código 4.13, a rotina principal cria uma *thread* (iniciada pela função *_callBack*2) e espera o final de sua execução. Observa-se que na linha 11, $EBP - 4$ referencia o *HANDLE* desta *thread*; o mesmo passado como argumento para o *WaitForSingleObject* na linha 15.

Código 4.13: Exemplo de uso *WaitForSingleObject* em *Threads*

```
1  _main:
2          (...)
3          mov     $0, 20(%esp)
4          mov     $0, 16(%esp)
5          mov     $1, 12(%esp)
6          mov     $_callBack2, 8(%esp)
7          mov     $0, 4(%esp)
8          mov     $0, (%esp)
9          call    _CreateThread@24
10         sub     $24, %esp
11         mov     %eax, -4(%ebp)
12         mov     $-1, 4(%esp)
13         mov     -4(%ebp), %eax
```

```
14          mov     %eax, (%esp)
15          call    _WaitForSingleObject@8
16          sub     $8, %esp
17          leave
18          ret
```

O segundo caso – *WaitForMultipleObjects* – foi introduzido ao mapeamento porque quando a rotina principal de um programa termina, todas as *threads* criadas por ela são destruídas. Porém, normalmente esse não é o comportamento desejado. Logo, costuma-se esperar que tais objetos terminem suas execuções antes de finalizar o programa por inteiro.

WaitForMultipleObjects permite que um *array* de *HANDLEs* seja passado, além de algumas outras configurações da espera. Seu mapeamento considera apenas um endereço inicial e um contador. A passagem de tal endereço inicial (*address* na linha 1 do Código 4.14) para o processo é feito de maneira semelhante ao do *HANDLE* do *WaitForSingleObject*. A partir dessas informações, considera-se que o conteúdo desses espaços de memória sejam preenchidos sequencialmente com os identificadores das *threads*. Recursivamente, o parâmetro contador (*count*) decresceo até que seja 1 – caso base da recursão. Passa-se a esperar, então, – através dos processos *WaitForSingleObject* – pela escrita no canal $ResumeThread\_channel[id]$ do inteiro 2 (terminação)– reservado para o *ExitThread*.

Código 4.14: Mapeamento de funções de espera

```
1  _WaitForMultipleObjects(address, count) =
2          ifa(count == 1) { _WaitForSingleObject(memory[HEAP][address]) }
3          else { (_WaitForSingleObject(memory[HEAP][address])
4                  || _WaitForMultipleObjects(address + 1, count - 1)) };
5
6  _WaitForSingleObject(id) =
7          ResumeThread_channel[id]?2 -> DoneWaiting -> Skip;
```

O mapeamento das esperas ignora o argumento relativo ao tempo, gerando sempre esperas infinitas. Além disso, na espera por mais de um objeto, somente os N primeiros itens são considerados, onde N é um parâmetro passado a esta função e mapeado pela entrada *count* no processo $\_WaitForMultilpeObjects$.

Quando, enfim, todas as *threads* sendo esperadas comunicaram seu término, o evento $DoneWaiting$ sincroniza e permite que o processo chamador retome o seu andamento.

É importante salientar que CSP# não implementa a sincronização múltipla de canais, como CSP_M. Com ela, quando mais de um processo está esperando a escrita, todos eles

são liberados. Ao invés disso, acontece uma escolha não-determinística sobre qual deles prossegue a execução; que é o comportamento desejado.

4.2.5 RCE 5: Paralelismo

A necessidade de haver paralelismo entre processos é analisada no código interno a cada *label*, onde se procuram chamadas às funções *CreateThread* ou *CreateMutex*.

Quando a primeira é detectada, a função passada como parâmetro (*callBack* no caso do Código 4.15) é mapeada de maneira diferente do usual. Um processo contendo o corpo da mesma ($callBack\_Body(id)$), mapeamento das instruções em si, é gerado e o original ($calBack(id)$) passa a ser esse corpo pré-fixado do evento $start.proc.id$. O processo na qual ocorre a chamada (processo *Label*) também passa a ter um corpo separado ($Label\_Body$) e é descrito como uma composição paralela de seu corpo com o processo que engloba a função referenciada no *CreateThread*. Dessa forma a execução do corpo da função fica dependendo da sincronização entre os eventos $start.proc.id$ nos diferentes processos. É definido um alfabeto para a paralelização para restringir as possibilidades do evento composto *start*, já que, inicialmente, ele recebe um inteiro qualquer.

Há um caso especial: quando o *CreateThread* está no escopo de uma *label* para o qual a execução incondicionalmente retorna, ou seja, há uma instrução *jmp* para esta ou alguma *label* anterior que possa fazer o *CreateThread* ser chamado mais de uma vez. Para esta situação – normalmente quando há uma chamada dentro de um laço *for* ou *while* no código fonte – considera-se que o modelo será executado uma primeira vez para definir quantas *threads* (quantas vezes se passa por aquele ponto) estão sendo criadas para que se coloque o número certo de processos em paralelo e seus respectivos identificadores.

Detectada a presença da segunda – *CreateMutex* –, o mesmo mecanismo de separação do corpo da função é aplicado. Porém, o processo ao qual esse corpo fica em paralelo é o pré-definido $Mutex$ mostrado no Código 4.15. O *CreateThread* e outras definições básicas foram omitidas pois já foram mostradas anteriormente.

O evento *End* é introduzido no final do escopo de quem cria o *mutex* para que, via sincronização desses eventos, ele termine sua execução com sucesso e não fique em recursão indefinidamente, gerando um estado em *deadlock* onde sempre se espera a escrita no canal $ResumeThread\_channel$.

Código 4.15: Mapeamento do paralelismo

```
1  #define callback_Proc 1;
2  #define MAX_THREADS 1;
3  channel LockMutex 0;
4  channel ReleaseMutex 0;
5  channel ResumeThread_channel[TOTALTHREADS] 0;
6  Mutex() =
7          LockMutex?id -> ReleaseMutex?id -> Mutex()
8          [] End -> Skip;
9
10 #alphabet Label_Body { proc:{callback_Proc}; thread_id:{1..MAX_THREADS}
11                        @ start.proc.thread_id, End };
12
13 Label() = Label_Body() || Mutex() || callBack(1);
14 Label_Body() =
15         a -> b -> _CreateThread(callback_Proc, 0);
16         c -> d -> call_CreateMutex ->
17         e -> f -> End -> Skip;
18
19 callBack(id) = start.callback_Proc.id -> callBack_Body(id); _ExitThread(id):
20 callBack_Body(id) = x -> y -> z -> Skip;
21
22 _ExitThread(id) =        ResumeThread_channel[id]!2 -> Skip;
```

4.3 Regras de Mapeamento de Instruções

Esta seção apresenta as regras de mapeamento das instruções. Apesar de baseado na estrutura proposta por Lima (2011), foi considerado desnecessária a presença de regras de nomenclatura. Já que o próprio compilador nomeia os componentes do *Assembly* de maneira razoável, decidiu-se por usar esses nomes de maneira direta. As seguintes Regras de Instruções (RI) são definidas.

4.3.1 RI 1: *Labels*

As *labels* que aparecem no código são mapeadas em um processo de nome equivalente. Há uma exceção: quando essas *labels* são usadas como rotinas de início de *threads* – passando a valer a RCE 5. São ignoradas terminações de nomes de *labels* decoradas pelo compilador com alguma informação. Portanto, *_soma*@8 gera um processo de nome *_soma*.

O processo mapeado recebe um identificador fictício inteiro não negativo ($Z+ = 0, 1, 2, 3, ...$) e único da *thread* na qual está rodando. Considera-se que a rotina principal do programa é uma linha de execução que ostenta o identificador 0 (zero). A partir dela, as próximas *threads* criadas obterão os identificadores consecutivos.

4.3.2 RI 2: Operações Aritméticas

O mapeamento das instruções aritméticas é bastante simples, de maneira que essas são apenas traduzidas para operações sobre dados (forma imperativa interna aos eventos de CSP#) e adequadas ao contexto deste trabalho. A Tabela 6 mostra como é feito esse mapeamento considerando que as instruções pertencem à *label* *_main* e à *thread* principal ($MAIN$).

Tabela 6: Mapeamento de Instruções Aritméticas Básicas

Instrução	Mapeamento
add $1, %eax	_main_add { eax[MAIN] = eax[MAIN] + 1 }
sub $1, %eax	_main_sub { eax[MAIN] = eax[MAIN] + 1 }
inc %eax	_main_inc { eax[MAIN]++ }
dec %eax	_main_dec { eax[MAIN]-- }
imul %ebx %eax	_main_imul { eax[MAIN] = ebx[MAIN] * eax[MAIN] }

Embora a aplicação dessa regra seja direta, há algumas exceções. Sempre que as instruções *add* ou *sub* estiverem alterando o valor dos registradores de controle da pilha, EBP ou ESP – apesar de ser mais comum encontrá-las associadas ao ESP –, o valor inteiro do primeiro operando será dividido por 4 como na Tabela 7; já que cada 4 *bytes* estão mapeados em apenas uma posição da memória no modelo CSP.

Tabela 7: Exceções de *add* e *sub*

Instrução	Mapeamento
add $4, %esp	_main_add { esp[MAIN] = esp[MAIN] + 1 }
sub $8, %esp	_main_sub { esp[MAIN] = esp[MAIN] - 2 }

4.3.3 RI 3: Desvios Condicionais

Instruções de desvio condicional utilizam o registrador de *flags* do processador para tomar suas decisões. Internamente, após fazer a subtração dos dois operandos, é feito um teste lógico que combina *Carry Flag*, *Zero Flag*, *Overflow Flag*, e *Parity Flag* a depender da instrução, para descobrir se será ou não feito o desvio.

Para manter a fidelidade com o *Assembly*, o desvio condicional do modelo CSP também passa pelas duas fases: comparação (*cmp*) e desvio. Os resultados de comparações (subtrações dos operandos) são guardados em um vetor chamado *cmps* de tamanho igual à quantidade de *threads* possíveis. Então um desvio condicional no CSP é feito sobre esse

valor, como mostra a Tabela 8 considerando novamente que se está no escopo de *_main*. As reticências indicam que qualquer código que apareça depois do desvio condicional será inserida naquele ponto da mesma maneira como vinha sendo feito antes de entrar no escopo do *if-else*.

Tabela 8: Desvios Condicionais

Instrução	**Mapeamento**
je Label	**ifa** (cmps[id] == 0) { _main_then -> Label() } **else** { _main_else -> (...) }
jne Label	**ifa** (cmps[id] != 0) { _main_then -> Label() } **else** { _main_else -> (...) }
jl Label	**ifa** (cmps[id] < 0) { _main_then -> Label() } **else** { _main_else -> (...) }
jle Label	**ifa** (cmps[id] <= 0) { _main_then -> Label() } **else** { _main_else -> (...) }
jg Label	**ifa** (cmps[id] > 0) { _main_then -> Label() } **else** { _main_else -> (...) }
jge Label	**ifa** (cmps[id] >= 0) { _main_then -> Label() } **else** { _main_else -> (...) }

O uso de *ifa* ao invés de um *if* simples é devido ao fato de que o último gera não-determinismo no modelo. Em CSP#, a avaliação da condição de um if gera um evento interno (τ). Logo, se duas *threads* diferentes estiverem em momentos de avaliação de if, haverá uma situação onde dois eventos iguais (τ) levam a estados diferentes. Desta forma, o modelo CSP# seria não-determinístico. Já com o *ifa*, não se tem esse problema, pois a avaliação da condição é feita atomicamente com o primeiro evento do if ou do $else$.

Para que esta decisão não afetasse a paridade com o comportamento do código *Assembly*, um evento nomeado *_then* ou *_else* prefixado da *label* atual é sempre colocado como primeiro o evento após a comparação, como um evento *dummy*.

4.3.4 RI 4: Desvios Incondicionais

O desvio incondicional (*jmp*) é mapeado simplesmente pondo o processo da *label* referenciada imediatamente após o evento mapeadao da instrução anterior. Isso fará o processo atual passar a se comportar como o processo da *label*.

Apesar de não ser um desvio, o resultado do mapeamento é o mesmo para quando há uma mudança de *label* em meio ao código *Assembly* sem que antes haja um retorno ou desvio incondicional. Para efeito de mapeamento, é como se a declaração de uma nova *label* sob essas condições fosse tratado como um *jmp* para a mesma.

4.3.5 RI 5: Movimentação de Dados

A instrução *mov* apresenta uma boa quantidade de variações possíveis para a cópia de dados entre seus dois argumentos. A Tabela 9 mostra algumas delas com seus respectivos mapeamentos. É importante frisar que quando um registrador aparece entre parênteses seu conteúdo é tratado como um endereço e o que é realmente copiado é o dado apontado por ele.

Tabela 9: Instrução *mov*

Instrução	**Mapeamento**
mov $0, %eax	Label_mov { eax[id] = 0}
mov $1, (%esp)	Label_mov { memory[id][esp[id]] = 1 }
mov (%eax), %eax	Label_mov { eax[id] = memory[id][eax[id]] }
mov %esp, %ebp	Label_mov { ebp[id] = esp[id] }
mov %eax, _y	Label_mov { memory[HEAP][_y] = eax[id] }
mov %eax, _y+4	Label_mov { memory[HEAP][_y + 1] = eax[id] }

A menção direta a uma variável global, como *_y* – definida anteriormente –, é mapeada como um acesso direto ou deslocado à memória.

4.3.6 RI 6: Operações com a Pilha de Execução

Além das instruções que podem afetar as variáveis de controle da pilha de execução vistas até aquivistas até aqui, como o *add*, o *sub* e o *mov* por exemplo, existem outras mais especificamente relacionadas com esse tipo de operação. A Tabela 10 mostra como essas instruções são mapeadas para o CSP#.

Há uma peculiaridade no mapeamento da instrução *call*. Como foi dito no Capítulo 2, ela empilha o valor atual do PC e então muda seu valor para o da *label* argumento. A idéia inicial deste trabalho era de abstrair o gerenciamento do endereço da próxima instrução, pois isso seria desnecessário. Porém, como o *call* é executado após o empilhamento dos argumentos dessa função a ser chamada, imediatamente após o desvio, o ESP está, na verdade, apontando para o valor antigo do PC e não para um dos argumentos. Isso faz com que o acesso aos parâmetros precise considerar os 4 *bytes* do PC guardado na hora de calcular o deslocamento em cima de EBP. Portanto, apesar de não ser usado na prática, o valor (macro) *pc_dummy*, de valor convencionado em 999, é sempre empilhada no evento de *call* mapeado.

Já a instrução *ret*, também descrita anteriormente, desempilha o PC antigo, fazendo

a execução voltar para a instrução seguinte ao *call*. Portanto, apesar de não se fazer nada com esse antigo valor, também é necessário mapeá-la para garantir que o valor de ESP esteja apontando para o endereço (índice) correto. Além disso, ela é a única instrução que é mapeada terminando em $SKIP$, pois é sempre a última instrução a aparecer dentro de um *label*.

Tabela 10: Instruções que atuam sobre a pilha

Instrução	**Mapeamento**
push %ebp	Label_push.id { esp[id]--; memory[id][esp[id]] = ebp[id] }
pop %ebp	Label_push.id { ebp[id] = memory[id][esp[id]]; esp[id]++ }
call Label2	call_Label2.id { esp[id]–; memory[id][esp[id]] = pc_dummy } -> Label2(id)
leave	Label_leave.id { esp[id] = ebp[id]; ebp[id] = memory[id][esp[id]]; esp[id]++ }
ret	Label_ret.id { esp[id]++ } -> **Skip**

4.3.7 RI 7: Módulo de 3

Apesar de não serem facilmente notadas, algumas operações de alto nível são compiladas de forma a otimizar a sua execução por parte de um processador específico.

Essa ideia é bastante utilizada, por exemplo, em instruções de multiplicação e divisão. Já que, respectivamente, adições ou subtrações sucessivas tornariam o custo de processamento bastante alto.

A divisão por três, quando compilada, é substituída por uma multiplicação pelo número 1431655766 seguido de 31 *shifts* aritméticos para a direita em um registrador de 64 bits. Realizando-se este cálculo, é possível observar que os 32 bits à esquerda são o resultado da divisão.

Como não há suporte para inteiros de 64 bits em CSP#, todo o Código 4.16 é substituído pela operação $\_MOD3\_especial.id\{eax[id] = ecx[id] - ((ecx[id]/3) * 3)\}$ durante o mapeamento.

Essa substituição não afeta o resultado da análise, pois não há alterações nos registradores ESP e EBP, nem na memória *heap*.

Código 4.16: Especial módulo de 3

```
1   mov   $1431655766, %eax
2   imul  %ecx
3   mov   %ecx, %eax
4   sar   $31, %eax
5   sub   %eax, %edx
6   mov   %edx, %eax
7   mov   %eax, -4(%ebp)
8   mov   -4(%ebp), %edx
9   mov   %edx, %eax
10  add   %eax, %eax
11  add   %edx, %eax
12  sub   %eax, %ecx
13  mov   %ecx, %eax
```

4.4 Aplicações

4.4.1 Exemplo 1: Produtores/Consumidores

Nesta seção as regras definidas até o momento são utilizadas na prática para a verificação de dois programas com problemas clássicos de concorrência: Produtores/Consumidores e Jantar dos Filósofos.

4.4.1.1 Aplicação das Regras

Este problema consiste na coordenação do acesso de tarefas de comportamento simétrico a um *buffer* limitado compartilhado. A tarefa do produtor é de criar itens e adicioná-los ao *buffer*, enquanto que a do consumidor é ir buscá-los para usar uma única vez. Caso o *buffer* esteja cheio, deve-se esperar o consumo de um item para retomar a tarefa de produção dos itens. Já quando está vazio, o consumidor deve aguardar o preenchimento para continuar sua execução (MAZIERO, 2011).

A versão implementada como exemplo para o trabalho contém um erro de concorrência conhecido: a depender do escalonamento das tarefas, é possível que ambas entrem em estado de suspensão, de onde nunca mais tornarão a executar. Isso acontece, por exemplo, quando o *buffer* está vazio e o consumidor decide se suspender.

Porém, a decisão e a suspensão não são uma operação atômica. Se antes de ser suspendida a tarefa for escalonada para o do produtor, um item pode ser produzido e, com isso, se tentará acordar a tarefa do consumidor (que ainda está acordada). Como o produtor só acorda o consumidor nesse momento específico, quando voltar a ser executado, o consumidor se suspenderá. O *buffer*, então, ficará totalmente preenchido levando o produtor também a dormir.

O Código 4.17 mostra como foi implementada a função *produzir* e a rotina principal, enquanto o Código 4.18 mostra o *Assembly* obtido a partir da função *produzir*.

Código 4.17: Função produzir

```
#include <stdio.h>
#include <windows.h>
#define MAXITENS 3
#define IDX_CONSUMIDOR 0
#define IDX_PRODUTOR 1
int itens = 0;
HANDLE threads[2];

produzir(void* parametros){
        while(1){
         if(itens == MAXITENS)
                 SuspendThread(threads[IDX_PRODUTOR]);

         itens++;

         if(itens == 1)
          ResumeThread(threads[IDX_CONSUMIDOR]);
        }
}

int main(void){
        threads[IDX_CONSUMIDOR] = CreateThread(NULL, 0, (void*)consumir, NULL, 0, NULL);
        threads[IDX_PRODUTOR] = CreateThread(NULL, 0, (void*)produzir, NULL, 0, NULL);
        WaitForMultipleObjects(2, threads, TRUE, INFINITE);
}
```

Código 4.18: Produzir compilado

```
_produzir:
        push    %ebp
        mov     %esp, %ebp
        sub     $8, %esp
L2:
        cmp     $3, _itens
        jne     L4
        mov     _threads+4, %eax
        mov     %eax, (%esp)
        call    _SuspendThread@4
        sub     $4, %esp
L4:
        inc     _itens
        cmp     $1, _itens
        jne     L2
        mov     _threads, %eax
        mov     %eax, (%esp)
        call    _ResumeThread@4
        sub     $4, %esp
        jmp     L2
```

A esse *Assembly* foram aplicadas as regras listadas na Tabela 11, gerando o CSP# do Código 4.19, que é uma versão preliminar do processo, já que outros processos ainda podem interferir na tradução do mesmo para CSP#. Nesse caso em específico, a *main* realmente irá alterá-lo.

Tabela 11: Aplicação das regras em "produzir"

Linha	**Regra**
1	RI 1
2	RI 6
3	RI 5
4	RI 2
5	RI 1
6 e 7	RI 3
8 e 9	RI 5
10	RCE 3
11	Não mapeada
12	RI 1
13	RI 2
14 e 15	RI 3
16 e 17	RI 5
18	RCE 3
19	Não mapeada
20	RI 4

O mapeamento do consumidor se assemelha bastante ao do produtor e, por isso, não será exposto aqui, mas pode ser encontrado no Apêndice A.1. Passa-se então ao mapeamento da versão compilada da rotina principal. No Código 4.20, o intervalo fechado das linhas $[11, 20]$ não gera especificação alguma. Além dessas, as linhas $\{28, 37, 44\}$ que sucedem chamadas externas à API, também são descartadas.

À parte os mapeamentos similares aos aplicados na função *produzir*, a rotina do Código 4.20 utiliza as seguintes regras: RCE 2 nas linhas 27 e 36; RCE 5 para execução paralela das *threads*; RCE 4 na linha 43; RCE 1 devido às variáveis globais nas linhas $[1, 5]$ e 47 – como mostrado no Código 4.5.

O Código 4.21 mostra as mudanças no processo *_produzir* e *_main*, que passaram a ter um corpo e a serem paralelos no processo que os englobava originalmente. Como dito na RCE 5, é definido o alfabeto de paralelismo, limitando as possibilidades do inteiro recebido pelo evento composto *start*. É importante atentar para o fato de que o processo *_produzir* antes definido, passa a ser chamado *_produzir_Body* devido à regra RCE 2.

Código 4.19: Produzir em CSP#

```
_produzir(id) =
        _produzir_push.id {memory[id][esp[id]] = ebp[id]; esp[id]--} ->
        _produzir_mov.id {esp[id] = ebp[id]} ->
        _produzir_sub.id {esp[id] = esp[id] - 2} ->
        L2(id);

L2(id) =
        L2_cmpl.id { cmps[id] = 3 - memory[HEAP][_itens] } ->
        ifa (cmps[id] != 0)
        { L2_then.id -> L4(id) }
        else
        {
         L2_else.id ->
         L2_mov.id {eax[id] = memory[HEAP][_threads + 1]} ->
         _SuspendThread(eax[id])
         ;L4(id)
        };

L4(id) =
        L4_inc.id { memory[HEAP][_itens]++ } ->
        L4_cmp.id { cmps[id] = 1 - memory[HEAP][_itens] } ->
        ifa (cmps[id] != 0)
        { L4then.id -> L2(id) }
        else
        {
         L4else.id ->
         L4_mov.id {eax[id] = memory[HEAP][_threads]} ->
         _ResumeThread(eax[id])
         ;L2(id)
        };
```

A especificação completa (Código A.3) foi submetida à verificação pela ferramenta PAT. Sabe-se que o código fonte original deste exemplo foi desenvolvido de maneira que nunca deveria chegar ao fim – há um laço que deveria deixar as tarefas do produtor e do consumidor rodando para sempre. Deseja-se, portanto, que ele seja um programa interminável (*nonterminating*) e, é claro, com ausência de *deadlocks* (*deadlockfree*).

A Figura 5 mostra, que a verificação falhou em ambas as assertivas.

4.4.1.2 Análise da Especificação

Analisando com cuidado o Código 4.22, que contém o rastro dos eventos ocorridos até o acontecimento do *deadlock*, e considerando apenas os fatos mais relevantes chega-se ao motivo da falha. Lembrando que o número após o nome dos eventos é apenas o identificador da *thread* na qual estes atuam.

1. Consumidor se suspende ($setThreadState$.1);
2. Produtor produz e acorda consumidor ($ResumeThread\_channel$[1].1);
3. Consumidor consome e verifica que deve dormir ($L7\_cmpl$.1 e $L7\_else$.1);

Código 4.20: *Assembly* da rotina principal dos Produtores/Consumidores

```
1  .globl _itens
2          .bss
3          .align 4
4  _itens:
5          .space 4
6          .text
7  _main:
8          push    %ebp
9          mov     %esp, %ebp
10         sub     $40, %esp
11         and     $-16, %esp
12         mov     $0, %eax
13         add     $15, %eax
14         add     $15, %eax
15         shr     $4, %eax
16         sal     $4, %eax
17         mov     %eax, -4(%ebp)
18         mov     -4(%ebp), %eax
19         call    __alloca
20         call    ___main
21         mov     $0, 20(%esp)
22         mov     $0, 16(%esp)
23         mov     $0, 12(%esp)
24         mov     $_consumir, 8(%esp)
25         mov     $0, 4(%esp)
26         mov     $0, (%esp)
27         call    _CreateThread@24
28         sub     $24, %esp
29         mov     %eax, _threads
30         mov     $0, 20(%esp)
31         mov     $0, 16(%esp)
32         mov     $0, 12(%esp)
33         mov     $_produzir, 8(%esp)
34         mov     $0, 4(%esp)
35         mov     $0, (%esp)
36         call    _CreateThread@24
37         sub     $24, %esp
38         mov     %eax, _threads+4
39         mov     $-1, 12(%esp)
40         mov     $1, 8(%esp)
41         mov     $_threads, 4(%esp)
42         mov     $2, (%esp)
43         call    _WaitForMultipleObjects@16
44         sub     $16, %esp
45         leave
46         ret
47         .comm   _threads, 16    # 8
```

Código 4.21: Mapeamento da *main* dos Produtores/Consumidores

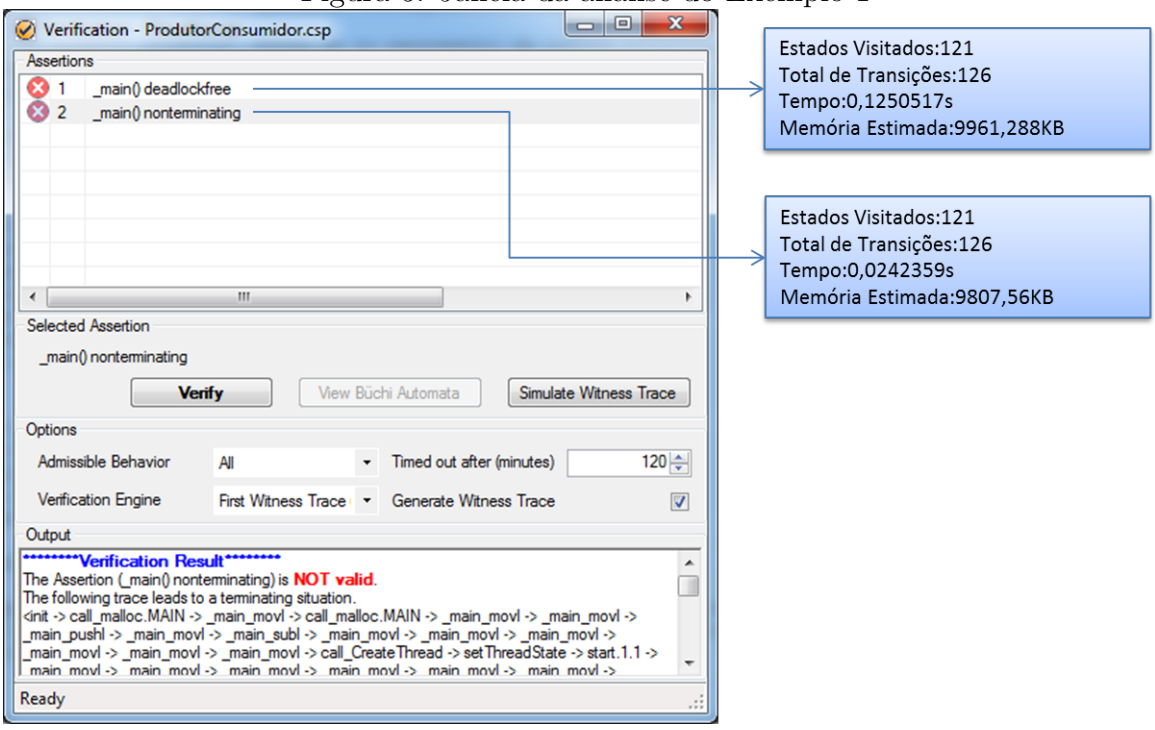
```
#define consumir_Proc 1;
#define produzir_Proc 2;

_main() = DefineGlobalVars(); _main_Body() || _consumir(1) || _produzir(2);

#alphabet _main_Body { proc:{produzir_Proc,consumir_Proc}; thread_id:{1..TOTALTHREADS}
                       @ start.proc.thread_id };
_main_Body() =
        _main_push { memory[MAIN][esp[MAIN]] = ebp[MAIN]; esp[MAIN]-- } ->
        _main_mov { ebp[MAIN] = esp[MAIN] } ->
        _main_sub { esp[MAIN] = esp[MAIN] - 10 } ->
        _main_mov { memory[MAIN][esp[MAIN] + 5] = 0 } ->
        _main_mov { memory[MAIN][esp[MAIN] + 4] = 0 } ->
        _main_mov { memory[MAIN][esp[MAIN] + 3] = 0 } ->
        _main_mov { memory[MAIN][esp[MAIN] + 2] = consumir_Proc } ->
        _main_mov { memory[MAIN][esp[MAIN] + 1] = 0 } ->
        _main_mov { memory[MAIN][esp[MAIN]] = 0  } ->
        _CreateThread(consumir_Proc, MAIN)
        ;_main_mov { memory[HEAP][_threads] = eax[MAIN] } ->
        _main_mov { memory[MAIN][esp[MAIN] + 5] = 0 } ->
        _main_mov { memory[MAIN][esp[MAIN] + 4] = 0 } ->
        _main_mov { memory[MAIN][esp[MAIN] + 3] = 0 } ->
        _main_mov { memory[MAIN][esp[MAIN] + 2] = produzir_Proc } ->
        _main_mov { memory[MAIN][esp[MAIN] + 1] = 0 } ->
        _main_mov { memory[MAIN][esp[MAIN]] = 0 } ->
        _CreateThread(produzir_Proc, MAIN)
        ;_main_mov { memory[HEAP][_threads + 1] = eax[MAIN] } ->
        _main_mov { memory[MAIN][esp[MAIN] + 3] = -1 } ->
        _main_mov { memory[MAIN][esp[MAIN] + 2] = 1 } ->
        _main_mov { memory[MAIN][esp[MAIN] + 1] = _threads } ->
        _main_mov { memory[MAIN][esp[MAIN]] = 2 } ->
        _WaitForMultipleObjects(_threads, 2)
        ;_main_leave{
                esp[MAIN] = ebp[MAIN];
                ebp[MAIN] = memory[MAIN][esp[MAIN]];
                esp[MAIN]--
        } -> Skip;

_produzir(id) =
        start.produzir_Proc.id ->
        _produzir_Body(id);
        _ExitThread(id);
```

Figura 5: Janela da análise do Exemplo 1



4. Produtor produz e "acorda" o consumidor suspenso ($[if((threadState[1] == 1))]$);

5. Consumidor se suspende($setThreadState.1$);

6. Produtor produz mais dois itens e dorme ($setThreadState.2$).

Código 4.22: *Trace* do *deadlock* no Exemplo 1

```
1  The Assertion (_main() deadlockfree) is NOT valid.
2  The following trace leads to a deadlock situation.
3  init -> call_malloc.MAIN -> _main_mov -> call_malloc.MAIN -> (...) -> _main_mov ->
4  call_CreateThread -> setThreadState -> start.1.1 -> _main_mov -> _main_mov -> (...) ->
5  _main_mov -> _main_mov -> call_CreateThread -> setThreadState -> start.2.2 ->
6  _main_mov -> _main_mov -> _main_mov -> _main_mov -> _main_mov -> _consumir_push.1 ->
7  _consumir_mov.1 -> _consumir_sub.1 -> L7_cmp.1 -> L7_else.1 -> L7_mov.1 ->
8  setThreadState.1 -> _produzir_push.2 -> _produzir_mov.2 -> _produzir_sub.2 ->
9  L2_cmp.2 -> L2_then.2 -> L4_inc.2 -> L4_cmp.2 -> L4else.2 -> L4_mov.2 ->
10 [if!((threadState[1] == 1))] -> setThreadState.1 -> ResumeThread_channel[1].1 ->
11 L9_dec.1 -> L9_cmp.1 -> L9_then.1 -> L7_cmp.1 -> L7_else.1 -> L2_cmp.2 -> L2_then.2 ->
12 L4_inc.2 -> L4_cmp.2 -> L4else.2 -> L4_mov.2 -> [if((threadState[1] == 1))] ->
13 L7_mov.1 -> setThreadState.1 -> L2_cmp.2 -> L2_then.2 -> L4_inc.2 -> L4_cmp.2 ->
14 L4then.2 -> L2_cmp.2 -> L2_then.2 -> L4_inc.2 -> L4_cmpl.2 -> L4then.2 -> L2_cmp.2 ->
15 L2_else.2 -> L2_mov.2 -> setThreadState.2
```

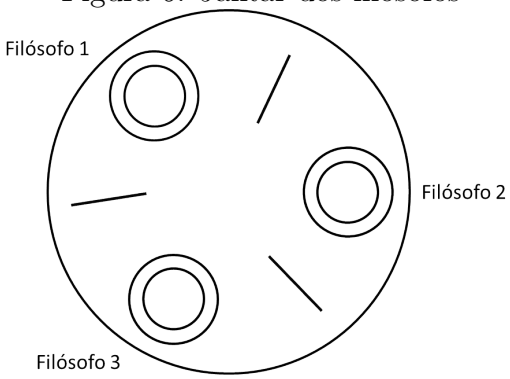
Ao fazer o mesmo para o caso da asserção *nonterminating*, percebe-se que o rastro é o idêntico ao anterior. Pois PAT considera que um *deadlock* é um caso de terminação.

4.4.2 Exemplo 2: Jantar dos Filósofos

4.4.2.1 Aplicação das regras

Mais um problema clássico, o Jantar dos Filósofos foi originalmente proposto por Djikstra e a ideia é que existem filósofos na mesa que alternam seu comportamento entre meditar e comer. Porém, para um deles comer é preciso que obtenha ambos os talheres, que são compartilhados, na sua esquerda e na sua direita. O Exemplo deste trabalho considerou o número de três filósofos, como na Figura 6. Convencionou-se que o garfo à esquerda de um filósofo tem o mesmo índice que ele e que o garfo à direta tem o índice conseguinte.

Figura 6: Jantar dos filósofos



A implementação deste exemplo deixa margem para a ocorrência de um *livelock*. É possível acontecer de os três filósofos pegarem o talher à sua esquerda, impedindo uns aos outros de obter ambos os talheres para comer. Não é possível, portanto, sair do *while* ao qual cada um deles está preso.

Código 4.23: Obtenção do talher direito

```
1  #define N 3
2  #define DIR(i) (((i)+1) %N)
3  void pegarTalherDir(int filNum){
4          int i = DIR(filNum);
5          BOOL pegou = 0;
6          while(pegou == FALSE){
7           WaitForSingleObject(Mutex, INFINITE);
8           if(garfos[i] == 0){
9            garfos[i] = 1;
10           pegou = TRUE;
11          }
12          ReleaseMutex(Mutex);
13         }}
```

O Código 4.23 mostra a função que representa pegar o talher da direita, cuja compilação gera gera o *Assembly* do Código 4.24. A aplicação das regras linha-a-linha é descrita na Tabela 12.

Código 4.24: *Assembly* da obtenção de talher direito

```
1  _pegarTalherDir:
2          push    %ebp
3          mov     %esp, %ebp
4          sub     $24, %esp
5          mov     8(%ebp), %ecx
6          inc     %ecx
7          mov     $1431655766, %eax
8          imul    %ecx
9          mov     %ecx, %eax
10         sar     $31, %eax
11         sub     %eax, %edx
12         mov     %edx, %eax
13         mov     %eax, -4(%ebp)
14         mov     -4(%ebp), %edx
15         mov     %edx, %eax
16         add     %eax, %eax
17         add     %edx, %eax
18         sub     %eax, %ecx
19         mov     %ecx, %eax
20         mov     %eax, -4(%ebp)
21         mov     $0, -8(%ebp)
22 L7:
23         cmp     $0, -8(%ebp)
24         jne     L6
25         mov     $-1, 4(%esp)
26         mov     _Mutex, %eax
27         mov     %eax, (%esp)
28         call    _WaitForSingleObject@8
29         sub     $8, %esp
30         mov     -4(%ebp), %eax
31         cmp     $0, _garfos(,%eax,4)
32         jne     L9
33         mov     -4(%ebp), %edx
34         mov     8(%ebp), %eax
35         inc     %eax
36         mov     %eax, _garfos(,%edx,4)
37         mov     $1, -8(%ebp)
38 L9:
39         mov     _Mutex, %eax
40         mov     %eax, (%esp)
41         call    _ReleaseMutex@4
42         sub     $4, %esp
43         jmp     L7
44 L6:
45         leave
46         ret
```

Tabela 12: Aplicação das regras em "pegarTalherDir"

Linha	**Regra**
1	RI 1
2	RI 6
3	RI 5
4	RI 2
5	RI 5
6	RI 2
7 a 19	RI 7
20 e 21	RI 5
22	RI 1
23 e 24	RI 3
25, 26 e 27	RI 5
28	RCE 4
29	Não mapeada
30	RI 5
31 e 32	RI 3
33 e 34	RI 5
35	RI 2
36 e 37	RI 5
38	RI 1
39 e 40	RI 5
41	RCE 5
42	Não mapeada
43	RI 4
44	RI 1
45 e 46	RI 6

As demais funções do programa possuem mapeamento similar ao feito para "pegarTalherDir" e, portanto, não são expostas nesta Seção.

Assim como no exemplo anterior, passa-se ao mapeamento da rotina principal (Código 4.25) – cujo *Assembly* pode ser visto no Apêndice A.2 –, onde são identificadas e aplicadas as regras de criação de *threads*, paralelismo (*mutex* e *threads*), variáveis globais, alocação de memória e espera.

Código 4.25: Rotina principal do problema dos filósofos

```
1  int garfos[N];
2  HANDLE threads[N];
3  HANDLE Mutex;
4  int main(){
5          int i;
6          int* f;
7          Mutex = CreateMutex(NULL, FALSE, NULL);
8          for(i = 0; i < N; i++){
9                  f = malloc(sizeof(int));
10                 *f = i;
11                 threads[i] = CreateThread(NULL,0, (void*)filosofo, f,0,NULL);
12         }
13         WaitForMultipleObjects(N, threads, TRUE, INFINITE); }
```

Após a compilação, a aplicação das regras de mapeamento deixa clara a interferência da *main* no procedimento de *label* $\_filosofos$ e, indiretamente, nos que são chamados internamente por ele: $\_pegarTalherEsq$, $\_pegarTalherDir$, $\_comer$, $\_devolverTalherEsq$ e $\_devolverTalherDir$.

4.4.2.2 Análise da Especificação

Finalizado o mapeamento, a especificação completa (Código A.6) foi verificada utilizando as mesmas asserções do exemplo passado: *deadlockfree* e *nonterminating*. Aqui, porém, além de se desejar que o programa seja livre de *deadlocks*, espera-se que o programa sempre termine.

Já foi dito anteriormente nesta Seção, que a implementação deste exemplo deixa margem para a ocorrência de *livelock*. Esse tipo de falha não é um estado de *deadlock* em CSP, então não é encontrada através da asserção *deadlockfree*.

Como já foi dito no Capítulo2, os *livelocks* em CSP estão associados a realizações sucessivas de transições internas, sem a comunicação de eventos visíveis em uma recursão infinita. Portanto, como o mapeamento não gera nenhum evento escondido, é esperado que a verificação se o processo é *divergencefree* seja válida.

Apesar de deixar o sistema indefinidamente em execução, esse *livelock* também não é encontrado via asserção *nonterminating*. Pois apesar de este caso específico ser interminável, em havendo ao menos um caso em que termine – seja por *deadlock* ou por finalização natural – o resultado será que a asserção é inválida.

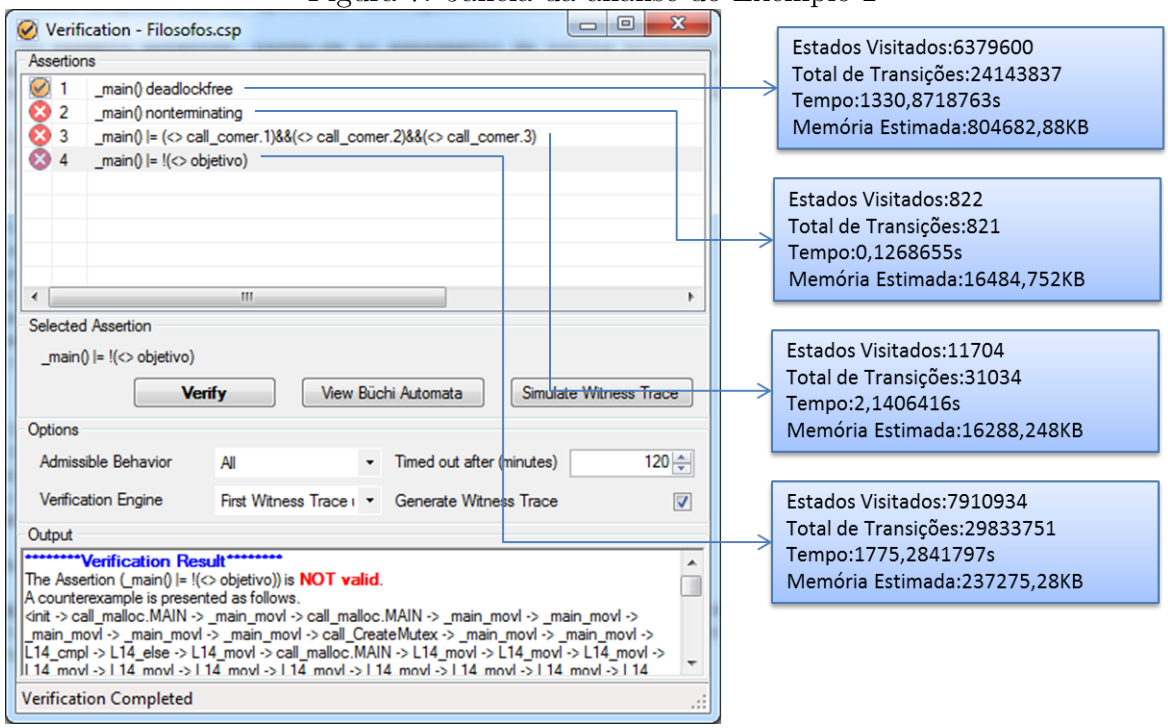
Há, ao menos, duas maneiras de detectar esta falha através de LTL. De ante-mão, sabe-se que todos os filósofos devem comer. Então, pode-se perguntar à ferramenta de verificação se não há nenhum estado futuro em que os três garfos estão com três diferentes filósofos. Outra alternativa é perguntar se, para todos os identificadores das *threads*, o evento *call_comer.id* é realizado em algum ponto de qualquer execução. Essas opções estã descritas no Código 4.26.

Código 4.26: Asserções LTL

```
1 #define objetivo (memory[HEAP][_garfos] == 1 &&
2         memory[HEAP][_garfos + 1] == 2 && memory[HEAP][_garfos + 2] == 3);
3 #assert _main() |= !(<> objetivo);
4 #assert _main() |= (<> call_comer.1) && (<> call_comer.2) && (<> call_comer.3);
```

A Figura 7 mostra o resultado das verificações feitas para este exemplo. Não foram encontrados rastros que levassem a *deadlocks*. Analisando o rastro provido pela verificação de *nonterminating*, percebe-se que este representa uma execução em que o programa terminou com sucesso.

Figura 7: Janela da análise do Exemplo 2



A veficação de alcance do estado definido por "objetivo" falha, ou seja, existe ao menos uma combinação de transições que leva àquele estado. O rastro mostra que cada um dos filósofos tem o garfo à sua esquerda. Também falhou a asserção de que todos as *threads* realizam o evento *call_comer.id* em algum ponto da execução.

5 Considerações Finais

Este capítulo apresenta as conclusões acerca dos resultados e de sua obtenção, bem como possíveis trabalhos futuros para melhoria de algumas das características observedas.

5.1 Conclusões

Neste trabalho de conclusão de curso foi proposto um conjunto de regras de mapeamento de código concorrente escrito em *Assembly* x86 para especificações CSP#. A aplicação dessas regras possibilita o uso do poder da verificação formal a partir do cálculo de processos e dá maior confiabilidade que os testes de *software* comuns, já que estes não investigam a fundo todas as possíveis combinações de escalonamento entre tarefas, podendo dar como correto um código passível de *deadlock* por exemplo.

Além disso, o fato de PAT permitir que o modelo resultante de um mapeamento também possa ser alvo, além das asserções usuais de CSP, de verificações LTL, como mostrado no segundo exemplo, garante um poder de verificação ainda maior.

Os objetivos descritos no Capítulo 1 foram atingidos e se apresentou uma solução candidata ao problema de pesquisa antes exposto, mas apesar dos bons resultados obtidos com o desenvolvimento deste estudo, algumas limitações estão presentes. Dentre elas, destacam-se:

1. Quantidade de exemplos: O pouco tempo disponível para a confecção do trabalho limitou a quantidade de exemplos a serem explorados. Isso implicou em uma menor quantidade de instruções mapeadas e em uma variedade de situações relativamente pequena em se falando de programação concorrente;

2. Especificidade do *Assembly* de origem: Apesar de a maioria das linguagens *Assembly* possuírem instruções de mesma finalidade, qualquer diferença, por menor que seja (sintaxe, nome de instrução, entre outros), pode levar à impossibilidade ou inconsis-

tência da aplicação das regras mostradas no Capítulo 4. Há, portanto, a limitação quanto ao alvo do mapeamento ser apenas o *Assembly* x86 para a plataforma Windows. Contudo, esta é uma plataforma de larga abrangência. O que motivou a escolha da mesma;

3. Tipo inteiro de CSP#: O inteiro de CSP# não permite que algumas operações comuns em código de máquina sejam reproduzidas, especialmente devido a problemas de *overflow* em uma multiplicação entre números muito grandes – maiores que 32 bits;

4. Natureza manual da aplicação das regras: As regras de mapeamento devem ser aplicadas manualmente. Então, ainda se corre o risco de uma falha humana interferir no resultado da análise. Além disso, devido à extensão dos códigos de máquina, é preciso bastante tempo para se concluir um trabalho manual de mapeamento;

5. Tamanho do modelo CSP gerado: É claro que este depende do código de entrada para aplicação das regras, mas o fato do mapeamento ser, a grosso modo, uma linha de código para uma operação em CSP# pode tornar a verificação um tanto lenta. Como há mais de um processo CSP em paralelo, a possibilidade de combinações no "escalonamento" torna-se muito grande, exigindo maior esforço computacional. O ideal seria detectar as regiões do código que podem gerar eventuais problemas de concorrência e restringir o paralelismo a esta(s) parte(s);

6. Difícil de detecção de *livelocks*: A detecção de *livelocks* não se tornou um procedimento automatizado. Portanto, o não conhecimento de suas possibilidades de ocorrência em certo código pode levar a crer em falsa ausência dos mesmos.

5.2 Trabalhos Futuros

Os seguintes trabalhos futuros são pertinentes, vistas as limitações desta pesquisa:

1. Realizar mapeamento de mais diferentes exemplos: Isso poderia gerar situações em que o conjunto de regras atual não seria capaz de mapear, gerando novas regras e, consequentemente, aumentando a abrangência da abordagem;

2. Criar uma ferramenta para aplicação automatizada: Dessa forma, os modelos gerados seriam mais confiáveis, pois não dependeriam da ação humana, além de aumentarem significativamente a velocidade de aplicação das regras;

3. Detectar regiões críticas no *Assembly*: Isso ocasionaria numa redução da quantidade de estados, pois paralelismo seria, então, limitado a essas regiões. Atualmente, vários dos estados considerados não geram condições de corrida e apenas tornam a verificação mais demorada;

4. Gerar verificações automáticas para desvios: Os desvios condicionais ou incondicionais podem gerar estados de *livelock* no programa. O ideal seria que o seu mapeamento gerasse uma verificação adicinal para esses casos, por exemplo, verificando se a comparação de um desvio condicional que pule para uma *label* anterior à corrente permite que se saia desse laço em algum ponto no futuro.

Referências

AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. [S.l.]: Addison Wesley, 2006. Hardcover. ISBN 0321486811.

CARVALHO, G. et al. Analytical comparison of refinement checkers. In: . [S.l.: s.n.], 2011.

CHARETTE, R. N. This car runs on code. *IEEE Spectrum*, fev. 2009.

DAHLWEID, M. et al. Vcc: Contract-based modular verification of concurrent c. In: *ICSE Companion*. [S.l.]: IEEE, 2009. p. 429–430. ISBN 978-1-4244-3494-7.

DVORAK, D. L.; LYU, M. Nasa study on flight software complexity. *Jet Propulsion*, American Institute of Aeronautics and Astronautics, 1801 Alexander Bell Dr., Suite 500 Reston VA 20191-4344 USA, p. 264, 1996.

FISHER, M. *An Introduction to Practical Formal Methods Using Temporal Logic*. [S.l.]: John Wiley & Sons, 2011. ISBN 9780470027882.

FSE. *FDR2 User Manual. Formal Systems (Europe) Ltd.* 9. ed. [S.l.], out. 2010.

HENNESSY, J. L.; PATTERSON, D. A.; LARUS, J. R. *Organização e Projeto de Computadores: A interface Hardware/Software*. 2. ed. Rio de Janeiro: Morgan Kaufmann, 2000. Tradução de Nery Machado Filho.

HOARE, C. A. R. *Communicating Sequential Processes*. [S.l.]: Prentice-Hall, 1985. ISBN 0-13-153271-5.

JURIC, Z.; REICHELT, S.; KOFLER, K. *The GNU Assembler*. [S.l.], 2003. Disponível em: <http://tigcc.ticalc.org/doc/gnuasm.html>.

KLEINE, M. et al. Llvm2csp: Extracting csp models from concurrent programs. In: BOBARU, M. G. et al. (Ed.). *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. [S.l.]: Springer, 2011. (Lecture Notes in Computer Science, v. 6617), p. 500–505. ISBN 978-3-642-20397-8.

LATTNER, C.; ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. Washington, DC, USA: IEEE Computer Society, 2004. (CGO '04), p. 75–. ISBN 0-7695-2102-9.

LIMA, H. L. C. de O. Análise formal de código java concorrente. In: *Trabalho de Conclusão de Curso*. Recife - PE: Engenharia da Computação - Universidade de Pernambuco, 2011.

MAUS, S.; MOSKAL, M.; SCHULTE, W. Vx86: x86 assembler simulated in c powered by automated theorem proving. In: *IN 12TH INTERNATIONAL CONFERENCE ON ALGEBRAIC METHODOLOGY AND SOFTWARE TECHNOLOGY (AMAST 2008), LNCS 5140*. [S.l.: s.n.], 2008.

MAZIERO, C. A. *Sistemas Operacionais*. [s.n.], 2011. Cap. 4. Disponível em: <http://dainf.ct.utfpr.edu.br/∼maziero/doku.php/so:livro_de_sistemas_operacionais>.

ROSCOE, A. W.; HOARE, C. A. R.; BIRD, R. *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997. ISBN 0136744095.

SCHNEIDER, S. *Concurrent and Real Time Systems: The CSP Approach*. 1st. ed. New York, NY, USA: John Wiley & Sons, Inc., 1999. ISBN 0471623733.

SUN, J.; LIU, Y.; DONG, J. *PAT: User Manual and Tutorial v3.4*. [S.l.], 2011.

WING, J. M. A specifier's introduction to formal methods. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 23, n. 9, p. 8–23, set. 1990. ISSN 0018-9162.

APÊNDICE A – Códigos-fonte e Mapeamentos dos Exemplos

Os códigos *Assembly* apresentam algumas instruções com nomeação diferente do restante do trabalho, pois são transcrições diretas dos arquivos gerados para uma maquina alvo que usa a versão de 64 bits do Windows.

A.1 Exemplo 1: Produtores e Consumidores

A.1.1 Código fonte em C

Código A.1: Produtores e Consumidores Completo

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <windows.h>
4  #define MAXITENS 3
5  #define IDX_CONSUMIDOR 0
6  #define IDX_PRODUTOR 1
7  int itens = 0;
8  HANDLE threads[2];
9
10 produzir(void* parametros){
11  while(1){
12    if(itens == MAXITENS)
13     SuspendThread(threads[IDX_PRODUTOR]);
14    itens++;
15    if(itens == 1)
16     ResumeThread(threads[IDX_CONSUMIDOR]);
17  }
18 }
19 void consumir(void* parametros){
20  while(1){
21    if(itens == 0)
22     SuspendThread(threads[IDX_CONSUMIDOR]);
23    itens--;
```

```
24      if(itens == MAXITENS - 1)
25       ResumeThread(threads[IDX_PRODUTOR]);
26    }
27  }
28  int main(void){
29    threads[IDX_CONSUMIDOR] = CreateThread(NULL, 0, (void*)consumir, NULL, 0, NULL);
30    threads[IDX_PRODUTOR] = CreateThread(NULL, 0, (void*)produzir, NULL, 0, NULL);
31    WaitForMultipleObjects(2, threads, TRUE, INFINITE);
32  }
```

A.1.2 *Assembly* x86

A compilação do Código A.1 a partir do comando "gcc -S ProdutorConsumidor.c" resulta no x86 seguinte:

Código A.2: x86 Produtores e Consumidores

```
1           .file   "ProdutorConsumidor.c"
2   .globl _itens
3           .bss
4           .align 4
5   _itens:
6           .space 4
7           .text
8   .globl _produzir
9           .def    _produzir;      .scl    2;      .type   32;     .endef
10  _produzir:
11          pushl   %ebp
12          movl    %esp, %ebp
13          subl    $8, %esp
14  L2:
15          cmpl    $3, _itens
16          jne     L4
17          movl    _threads+4, %eax
18          movl    %eax, (%esp)
19          call    _SuspendThread@4
20          subl    $4, %esp
21  L4:
22          incl    _itens
23          cmpl    $1, _itens
24          jne     L2
25          movl    _threads, %eax
26          movl    %eax, (%esp)
27          call    _ResumeThread@4
28          subl    $4, %esp
29          jmp     L2
30  .globl _consumir
31          .def    _consumir;      .scl    2;      .type   32;     .endef
32  _consumir:
33          pushl   %ebp
34          movl    %esp, %ebp
35          subl    $8, %esp
```

```
36 L7:
37          cmpl    $0, _itens
38          jne     L9
39          movl    _threads, %eax
40          movl    %eax, (%esp)
41          call    _SuspendThread@4
42          subl    $4, %esp
43 L9:
44          decl    _itens
45          cmpl    $2, _itens
46          jne     L7
47          movl    _threads+4, %eax
48          movl    %eax, (%esp)
49          call    _ResumeThread@4
50          subl    $4, %esp
51          jmp     L7
52          .def    ___main; .scl 2; .type 32; .endef
53 .globl _main
54          .def    _main; .scl 2; .type 32; .endef
55 _main:
56          pushl   %ebp
57          movl    %esp, %ebp
58          subl    $40, %esp
59          andl    $-16, %esp
60          movl    $0, %eax
61          addl    $15, %eax
62          addl    $15, %eax
63          shrl    $4, %eax
64          sall    $4, %eax
65          movl    %eax, -4(%ebp)
66          movl    -4(%ebp), %eax
67          call    __alloca
68          call    ___main
69          movl    $0, 20(%esp)
70          movl    $0, 16(%esp)
71          movl    $0, 12(%esp)
72          movl    $_consumir, 8(%esp)
73          movl    $0, 4(%esp)
74          movl    $0, (%esp)
75          call    _CreateThread@24
76          subl    $24, %esp
77          movl    %eax, _threads
78          movl    $0, 20(%esp)
79          movl    $0, 16(%esp)
80          movl    $0, 12(%esp)
81          movl    $_produzir, 8(%esp)
82          movl    $0, 4(%esp)
83          movl    $0, (%esp)
84          call    _CreateThread@24
85          subl    $24, %esp
86          movl    %eax, _threads+4
87          movl    $-1, 12(%esp)
88          movl    $1, 8(%esp)
89          movl    $_threads, 4(%esp)
```

```
90          movl    $2, (%esp)
91          call    _WaitForMultipleObjects@16
92          subl    $16, %esp
93          leave
94          ret
95          .comm   _threads, 16    # 8
```

A.1.3 CSP#

A aplicação das regras de mapeamento no Código A.2 gera a seguinte especificação CSP#:

Código A.3: CSP# Produtores e Consumidores

```
1  #define MEM_SIZE 30;
2  #define MEM_LAST_INDEX 29;
3  #define MAIN 0;
4  #define MAX_THREADS 2;
5  #define TOTALTHREADS 3;
6  #define HEAP TOTALTHREADS;
7  #define consumir_Proc 1;
8  #define produzir_Proc 2;
9  #define pc_dummy 999;
10
11 //Memória
12 var memory[TOTALTHREADS + 1][MEM_SIZE];
13 var threadState = [1, 0(MAX_THREADS)];
14 channel ResumeThread_channel[TOTALTHREADS] 0;
15 var cmps[TOTALTHREADS];
16
17 //Contador de criação de threads
18 var current_id = 0;
19 //Contador de alocação na heap
20 var current_heap = 0;
21
22 //Registradores
23 var eax[TOTALTHREADS];
24 var ebp = [MEM_LAST_INDEX(TOTALTHREADS)];
25 var esp = [MEM_LAST_INDEX(TOTALTHREADS)];
26
27 //Variáveis das seções bss e data
28 var _itens;
29 var _threads;
30
31 _CreateThread (proc, creatorId) =
32  atomic{
33   call_CreateThread {
34    current_id++;
35    esp[current_id] = esp[current_id] - 1; //sub 4, %esp
36    // Passagem 3º do parâmetro do CreateThread para função de CallBack
37    memory[current_id][esp[current_id]] = memory[creatorId][esp[creatorId] + 3];
38    esp[current_id]--; memory[current_id][esp[current_id]] = pc_dummy; // push %PC
```

```
39      eax[creatorId] = current_id; // Coloca id no retorno
40    } -> setThreadState { threadState[(eax[creatorId])] = 1 } ->
41    start.proc.(eax[creatorId]) -> Skip
42   };
43
44  _ExitThread(id) =
45   ResumeThread_channel[id]!2 -> Skip;
46
47  Malloc(id, size) =
48   call_malloc.id {
49    eax[id] = current_heap;
50    current_heap = current_heap + size/4;
51   } -> Skip;
52
53  DefineGlobalVars() =
54   Malloc(MAIN, 4)
55   ; _main_movl { _itens = eax[MAIN] } ->
56   Malloc(MAIN, 8)
57   ; _main_movl { _threads = eax[MAIN] } ->
58   _main_movl { eax[MAIN] = 0 } -> Skip;
59
60  _ResumeThread(id) =
61   atomic {
62    if(threadState[id] == 1) { Skip }
63    else {
64      setThreadState.id { threadState[id] = 1 } ->
65      ResumeThread_channel[id]!1 -> Skip
66    }
67   };
68
69  _SuspendThread(id) =
70   atomic{ setThreadState.id { threadState[id] = 0 } ->
71    ResumeThread_channel[id]?1 -> Skip
72   };
73
74  _WaitForMultipleObjects(address, count) =
75   ifa(count == 1) { _WaitForSingleObject(memory[HEAP][address]) }
76   else { (_WaitForSingleObject(memory[HEAP][address])
77          || _WaitForMultipleObjects(address + 1, count - 1)) };
78
79  _WaitForSingleObject(id) =
80   ResumeThread_channel[id]?2 -> DoneWaiting -> Skip;
81
82  #alphabet _produzir { thread_id:{1..TOTALTHREADS} @ start.produzir_Proc.thread_id };
83  _produzir(id) =
84   start.produzir_Proc.id ->
85   _produzir_Body(id);
86   _ExitThread(id);
87
88  _produzir_Body(id) =
89   _produzir_push.id {memory[id][esp[id]] = ebp[id]; esp[id]--} ->
90   _produzir_movl.id {esp[id] = ebp[id]} ->
91   _produzir_subl.id {esp[id] = esp[id] - 2} ->
92   L2(id);
```

```
93
94  L2(id) =
95   L2_cmpl.id { cmps[id] = 3 - memory[HEAP][_itens] } ->
96   ifa (cmps[id] != 0)
97   { L2_then.id -> L4(id) }
98   else
99   {
100   L2_else.id ->
101   L2_movl.id {eax[id] = memory[HEAP][_threads + 1]} ->
102   _SuspendThread(eax[id])
103   ;L4(id)
104  };
105
106 L4(id) =
107  L4_incl.id { memory[HEAP][_itens]++ } ->
108  L4_cmpl.id { cmps[id] = 1 - memory[HEAP][_itens] } ->
109  ifa (cmps[id] != 0)
110  { L4then.id -> L2(id) }
111  else
112  {
113   L4else.id ->
114   L4_movl.id {eax[id] = memory[HEAP][_threads]} ->
115   _ResumeThread(eax[id])
116   ;L2(id)
117  };
118
119 #alphabet _consumir { thread_id:{1..TOTALTHREADS} @ start.consumir_Proc.thread_id };
120 _consumir(id) =
121  start.consumir_Proc.id ->
122  _consumir_Body(id);
123  _ExitThread(id);
124
125 _consumir_Body(id) =
126  _consumir_push.id {memory[id][esp[id]] = ebp[id]; esp[id]--} ->
127  _consumir_movl.id {esp[id] = ebp[id]} ->
128  _consumir_subl.id {esp[id] = esp[id] - 2} ->
129  L7(id);
130
131 L7(id) =
132  L7_cmpl.id { cmps[id] = 0 - memory[HEAP][_itens]} ->
133  ifa (cmps[id] != 0)
134  { L7_then.id -> L9(id) }
135  else
136  {
137   L7_else.id ->
138   L7_movl.id {eax[id] = memory[HEAP][_threads]} ->
139   _SuspendThread(eax[id])
140   ;L9(id)
141  };
142
143 L9(id) =
144  L9_decl.id { memory[HEAP][_itens]-- } ->
145  L9_cmpl.id { cmps[id] = 2 - memory[HEAP][_itens] } ->
146  ifa (cmps[id] != 0)
```

```
147  { L9_then.id -> L7(id) }
148  else
149  {
150   L9_else.id ->
151   L9_movl.id {eax[id] = memory[HEAP][_threads + 1]} ->
152   _ResumeThread(eax[id])
153   ;L7(id)
154  };
155
156 _main() = DefineGlobalVars(); _main_Body() || _consumir(1) || _produzir(2);
157
158 #alphabet _main_Body { proc:{produzir_Proc,consumir_Proc};
159          thread_id:{1..TOTALTHREADS} @ start.proc.thread_id };
160 _main_Body() =
161  _main_pushl { memory[MAIN][esp[MAIN]] = ebp[MAIN]; esp[MAIN]-- } ->
162  _main_movl { ebp[MAIN] = esp[MAIN] } ->
163  _main_subl { esp[MAIN] = esp[MAIN] - 10 } ->
164  _main_movl { memory[MAIN][esp[MAIN] + 5] = 0 } ->
165  _main_movl { memory[MAIN][esp[MAIN] + 4] = 0 } ->
166  _main_movl { memory[MAIN][esp[MAIN] + 3] = 0 } ->
167  _main_movl { memory[MAIN][esp[MAIN] + 2] = consumir_Proc } ->
168  _main_movl { memory[MAIN][esp[MAIN] + 1] = 0 } ->
169  _main_movl { memory[MAIN][esp[MAIN]] = 0  } ->
170  _CreateThread(consumir_Proc, MAIN)
171  ;_main_movl { memory[HEAP][_threads] = eax[MAIN] } ->
172  _main_movl { memory[MAIN][esp[MAIN] + 5] = 0 } ->
173  _main_movl { memory[MAIN][esp[MAIN] + 4] = 0 } ->
174  _main_movl { memory[MAIN][esp[MAIN] + 3] = 0 } ->
175  _main_movl { memory[MAIN][esp[MAIN] + 2] = produzir_Proc } ->
176  _main_movl { memory[MAIN][esp[MAIN] + 1] = 0 } ->
177  _main_movl { memory[MAIN][esp[MAIN]] = 0 } ->
178  _CreateThread(produzir_Proc, MAIN)
179  ;_main_movl { memory[HEAP][_threads + 1] = eax[MAIN] } ->
180  _main_movl { memory[MAIN][esp[MAIN] + 3] = -1 } ->
181  _main_movl { memory[MAIN][esp[MAIN] + 2] = 1 } ->
182  _main_movl { memory[MAIN][esp[MAIN] + 1] = _threads } ->
183  _main_movl { memory[MAIN][esp[MAIN]] = 2 } ->
184  _WaitForMultipleObjects(_threads, 2)
185  ;_main_leave{
186   esp[MAIN] = ebp[MAIN];
187   ebp[MAIN] = memory[MAIN][esp[MAIN]];
188   esp[MAIN]--
189  } -> Skip;
190 #assert _main() deadlockfree;
191 #assert _main() nonterminating;
```

A.2 Exemplo 2: O Jantar dos Filósofos

A.2.1 Código fonte em C

Código A.4: Jantar dos Filósofos Completo

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#define N 3 //N filósofos
#define DIR(i) (((i)+1) %N) //Garfo direita
int garfos[N];
HANDLE threads[N];
HANDLE Mutex;
void comer(int filNum){ }
void pegarTalherEsq(int filNum){
 int i = filNum;
 BOOL pegou = 0;
 while(pegou == FALSE){
  WaitForSingleObject(Mutex, INFINITE);
  if(garfos[i] == 0){
   garfos[i] = filNum + 1;
   pegou = TRUE;
  }
  ReleaseMutex(Mutex);
 }
}
void pegarTalherDir(int filNum){
 int i = DIR(filNum);
 BOOL pegou = 0;
 while(pegou == FALSE){
  WaitForSingleObject(Mutex, INFINITE);
  if(garfos[i] == 0){
   garfos[i] = filNum + 1;
   pegou = TRUE;
  }
  ReleaseMutex(Mutex);
 }
}
void devolverTalherEsq(int filNum){
 int i = filNum;
 garfos[i] = 0;
}
void devolverTalherDir(int filNum){
 int i = DIR(filNum);
 garfos[i] = 0;
}
void filosofo(void* parametros){
 //Número do filósofo passado como parametro para a thread
 int* filNum = (int*)parametros;

 pegarTalherEsq(*filNum);
 pegarTalherDir(*filNum);
 comer(*filNum);
 devolverTalherEsq(*filNum);
 devolverTalherDir(*filNum);
}
int main(){
 int i;
```

```
54  int* f;
55  Mutex = CreateMutex(NULL, FALSE, NULL);
56  for(i = 0; i < N; i++){
57   f = malloc(sizeof(int));
58   *f = i;
59   threads[i] = CreateThread(NULL,0, (void*)filosofo, f,0,NULL);
60  }
61  WaitForMultipleObjects(N, threads, TRUE, INFINITE);
62 }
```

A.2.2 *Assembly* x86

A compilação do Código A.4 a partir do comando "gcc -S JantarFilosofos.c" resulta no x86 seguinte:

Código A.5: x86 Jantar dos Filósofos

```
1           .file   "JantarFilosofos.c"
2           .text
3  .globl _comer
4           .def    _comer; .scl    2;      .type   32;     .endef
5  _comer:
6           pushl   %ebp
7           movl    %esp, %ebp
8           popl    %ebp
9           ret
10 .globl _pegarTalherEsq
11          .def    _pegarTalherEsq;        .scl    2;      .type   32;     .endef
12 _pegarTalherEsq:
13          pushl   %ebp
14          movl    %esp, %ebp
15          subl    $24, %esp
16          movl    8(%ebp), %eax
17          movl    %eax, -4(%ebp)
18          movl    $0, -8(%ebp)
19 L3:
20          cmpl    $0, -8(%ebp)
21          jne     L2
22          movl    $-1, 4(%esp)
23          movl    _Mutex, %eax
24          movl    %eax, (%esp)
25          call    _WaitForSingleObject@8
26          subl    $8, %esp
27          movl    -4(%ebp), %eax
28          cmpl    $0, _garfos(,%eax,4)
29          jne     L5
30          movl    -4(%ebp), %edx
31          movl    8(%ebp), %eax
32          incl    %eax
33          movl    %eax, _garfos(,%edx,4)
34          movl    $1, -8(%ebp)
35 L5:
```

```
36          movl    _Mutex, %eax
37          movl    %eax, (%esp)
38          call    _ReleaseMutex@4
39          subl    $4, %esp
40          jmp     L3
41  L2:
42          leave
43          ret
44  .globl _pegarTalherDir
45          .def    _pegarTalherDir;        .scl    2;      .type   32;     .endef
46  _pegarTalherDir:
47          pushl   %ebp
48          movl    %esp, %ebp
49          subl    $24, %esp
50          movl    8(%ebp), %ecx
51          incl    %ecx
52          movl    $1431655766, %eax
53          imull   %ecx
54          movl    %ecx, %eax
55          sarl    $31, %eax
56          subl    %eax, %edx
57          movl    %edx, %eax
58          movl    %eax, -4(%ebp)
59          movl    -4(%ebp), %edx
60          movl    %edx, %eax
61          addl    %eax, %eax
62          addl    %edx, %eax
63          subl    %eax, %ecx
64          movl    %ecx, %eax
65          movl    %eax, -4(%ebp)
66          movl    $0, -8(%ebp)
67  L7:
68          cmpl    $0, -8(%ebp)
69          jne     L6
70          movl    $-1, 4(%esp)
71          movl    _Mutex, %eax
72          movl    %eax, (%esp)
73          call    _WaitForSingleObject@8
74          subl    $8, %esp
75          movl    -4(%ebp), %eax
76          cmpl    $0, _garfos(,%eax,4)
77          jne     L9
78          movl    -4(%ebp), %edx
79          movl    8(%ebp), %eax
80          incl    %eax
81          movl    %eax, _garfos(,%edx,4)
82          movl    $1, -8(%ebp)
83  L9:
84          movl    _Mutex, %eax
85          movl    %eax, (%esp)
86          call    _ReleaseMutex@4
87          subl    $4, %esp
88          jmp     L7
89  L6:
```

```
90          leave
91          ret
92  .globl _devolverTalherEsq
93          .def    _devolverTalherEsq;     .scl    2;      .type   32;     .endef
94  _devolverTalherEsq:
95          pushl   %ebp
96          movl    %esp, %ebp
97          subl    $4, %esp
98          movl    8(%ebp), %eax
99          movl    %eax, -4(%ebp)
100         movl    -4(%ebp), %eax
101         movl    $0, _garfos(,%eax,4)
102         leave
103         ret
104 .globl _devolverTalherDir
105         .def    _devolverTalherDir;     .scl    2;      .type   32;     .endef
106 _devolverTalherDir:
107         pushl   %ebp
108         movl    %esp, %ebp
109         subl    $4, %esp
110         movl    8(%ebp), %ecx
111         incl    %ecx
112         movl    $1431655766, %eax
113         imull   %ecx
114         movl    %ecx, %eax
115         sarl    $31, %eax
116         subl    %eax, %edx
117         movl    %edx, %eax
118         movl    %eax, -4(%ebp)
119         movl    -4(%ebp), %edx
120         movl    %edx, %eax
121         addl    %eax, %eax
122         addl    %edx, %eax
123         subl    %eax, %ecx
124         movl    %ecx, %eax
125         movl    %eax, -4(%ebp)
126         movl    -4(%ebp), %eax
127         movl    $0, _garfos(,%eax,4)
128         leave
129         ret
130 .globl _filosofo
131         .def    _filosofo;      .scl    2;      .type   32;     .endef
132 _filosofo:
133         pushl   %ebp
134         movl    %esp, %ebp
135         subl    $8, %esp
136         movl    8(%ebp), %eax
137         movl    %eax, -4(%ebp)
138         movl    -4(%ebp), %eax
139         movl    (%eax), %eax
140         movl    %eax, (%esp)
141         call    _pegarTalherEsq
142         movl    -4(%ebp), %eax
143         movl    (%eax), %eax
```

```
144         movl    %eax, (%esp)
145         call    _pegarTalherDir
146         movl    -4(%ebp), %eax
147         movl    (%eax), %eax
148         movl    %eax, (%esp)
149         call    _comer
150         movl    -4(%ebp), %eax
151         movl    (%eax), %eax
152         movl    %eax, (%esp)
153         call    _devolverTalherEsq
154         movl    -4(%ebp), %eax
155         movl    (%eax), %eax
156         movl    %eax, (%esp)
157         call    _devolverTalherDir
158         leave
159         ret
160         .def    ___main; .scl 2; .type 32; .endef
161 .globl _main
162         .def    _main; .scl 2; .type 32; .endef
163 _main:
164         pushl   %ebp
165         movl    %esp, %ebp
166         pushl   %ebx
167         subl    $36, %esp
168         andl    $-16, %esp
169         movl    $0, %eax
170         addl    $15, %eax
171         addl    $15, %eax
172         shrl    $4, %eax
173         sall    $4, %eax
174         movl    %eax, -16(%ebp)
175         movl    -16(%ebp), %eax
176         call    __alloca
177         call    ___main
178         movl    $0, 8(%esp)
179         movl    $0, 4(%esp)
180         movl    $0, (%esp)
181         call    _CreateMutexA@12
182         subl    $12, %esp
183         movl    %eax, _Mutex
184         movl    $0, -8(%ebp)
185 L14:
186         cmpl    $2, -8(%ebp)
187         jg      L15
188         movl    $4, (%esp)
189         call    _malloc
190         movl    %eax, -12(%ebp)
191         movl    -12(%ebp), %edx
192         movl    -8(%ebp), %eax
193         movl    %eax, (%edx)
194         movl    -8(%ebp), %ebx
195         movl    $0, 20(%esp)
196         movl    $0, 16(%esp)
197         movl    -12(%ebp), %eax
```

```
198         movl    %eax, 12(%esp)
199         movl    $_filosofo, 8(%esp)
200         movl    $0, 4(%esp)
201         movl    $0, (%esp)
202         call    _CreateThread@24
203         subl    $24, %esp
204         movl    %eax, _threads(,%ebx,4)
205         leal    -8(%ebp), %eax
206         incl    (%eax)
207         jmp     L14
208 L15:
209         movl    $-1, 12(%esp)
210         movl    $1, 8(%esp)
211         movl    $_threads, 4(%esp)
212         movl    $3, (%esp)
213         call    _WaitForMultipleObjects@16
214         subl    $16, %esp
215         movl    -4(%ebp), %ebx
216         leave
217         ret
218         .comm   _garfos, 16     # 12
219         .comm   _threads, 16    # 12
220         .comm   _Mutex, 16      # 4
221         .def    _malloc;        .scl    3;      .type   32;     .endef
```

A.2.3 CSP#

A aplicação das regras de mapeamento no Código A.5 gera a seguinte especificação CSP#:

Código A.6: CSP# Jantar dos Filósofos

```
1  #define MEM_SIZE 30;
2  #define MEM_LAST_INDEX 29;
3  #define MAIN 0;
4  #define MAX_THREADS 3;
5  #define TOTALTHREADS 4;
6  #define HEAP TOTALTHREADS;
7  #define filosofo_Proc 1;
8  #define pc_dummy 999;
9
10 //Memória
11 var memory[TOTALTHREADS + 1][MEM_SIZE];
12 var threadState = [1, 0(MAX_THREADS)];
13 var cmps[TOTALTHREADS];
14
15 channel ResumeThread_channel[TOTALTHREADS] 0;
16
17 //Contador de criação de threads
18 var current_id = 0;
19 //Contador de alocação na heap
20 var current_heap = 0;
```

```
21
22 //Registradores
23 var esp = [MEM_LAST_INDEX(TOTALTHREADS)];
24 var ebp = [MEM_LAST_INDEX(TOTALTHREADS)];
25 var eax[TOTALTHREADS];
26 var ebx[TOTALTHREADS];
27 var ecx[TOTALTHREADS];
28 var edx[TOTALTHREADS];
29
30 //Variáveis das seções bss e data
31 var _garfos;
32 var _threads;
33 var _Mutex;
34
35 _CreateThread (proc, creatorId) =
36  atomic{
37   call_CreateThread {
38    current_id++;
39    esp[current_id] = esp[current_id] - 1; //sub 4, %esp
40    // Passagem 3º do parâmetro do CreateThread para função de CallBack
41    memory[current_id][esp[current_id]] = memory[creatorId][esp[creatorId] + 3];
42    esp[current_id]--; memory[current_id][esp[current_id]] = pc_dummy; // push %PC
43    eax[creatorId] = current_id; // Coloca id no retorno
44   } -> setThreadState { threadState[(eax[creatorId])] = 1 } ->
45   start.proc.(eax[creatorId]) -> Skip
46  };
47
48 _ExitThread(id) =
49  ResumeThread_channel[id]!2 -> Skip;
50
51 Malloc(id, size) =
52  call_malloc.id {
53   eax[id] = current_heap;
54   current_heap = current_heap + size/4;
55  } -> Skip;
56
57 DefineGlobalVars() =
58  Malloc(MAIN, 12)
59  ; _main_movl { _garfos = eax[MAIN]; } ->
60  Malloc(MAIN, 12)
61  ; _main_movl { _threads = eax[MAIN] } ->
62  _main_movl { eax[MAIN] = 0 } -> Skip;
63
64 _WaitForMultipleObjects(address, count) =
65  ifa(count == 1) { _WaitForSingleObject(memory[HEAP][address]) }
66  else { (_WaitForSingleObject(memory[HEAP][address])
67          || _WaitForMultipleObjects(address + 1, count - 1)) };
68
69 _WaitForSingleObject(id) =
70  ResumeThread_channel[id]?2 -> DoneWaiting -> Skip;
71
72 channel LockMutex 0;
73 channel ReleaseMutex 0;
74
```

```
75  Mutex() =
76   LockMutex?id -> ReleaseMutex?id -> Mutex()
77   [] End -> Skip;
78
79  _main() = _main_Body() || Mutex() || (|| x:{1..MAX_THREADS} @ _filosofo(x));
80
81  #alphabet _main_Body { proc:{filosofo_Proc};
82          thread_id:{1..MAX_THREADS} @ start.proc.thread_id, End };
83  _main_Body() =
84   DefineGlobalVars();
85   _main_movl{ memory[MAIN][esp[MAIN]-2] = 0 } ->
86   _main_movl{ memory[MAIN][esp[MAIN]-1] = 0 } ->
87   _main_movl{ memory[MAIN][esp[MAIN]] = 0 } ->
88   call_CreateMutex ->
89   _main_movl{ eax[MAIN] = _Mutex } ->
90   _main_movl{ memory[MAIN][ebp[MAIN] - 2] = 0 } ->
91   L14();
92
93  L14() =
94   L14_cmpl{ cmps[MAIN] = 2 - memory[MAIN][ebp[MAIN] - 2] } ->
95   ifa (cmps[MAIN] < 0)
96   {
97    L14_then-> L15()
98   }
99   else
100  {
101   L14_else ->
102   L14_movl { memory[MAIN][esp[MAIN]] = 4 } ->
103   Malloc(MAIN, 4)
104   ;L14_movl { memory[MAIN][ebp[MAIN] - 3] = eax[MAIN] } ->
105   L14_movl { edx[MAIN] = memory[MAIN][ebp[MAIN] - 3] } ->
106   L14_movl { eax[MAIN] = memory[MAIN][ebp[MAIN] - 2] } ->
107   L14_movl { memory[HEAP][edx[MAIN]] = eax[MAIN] } ->
108   L14_movl { ebx[MAIN] = memory[MAIN][ebp[MAIN] - 2] } ->
109   L14_movl { memory[MAIN][esp[MAIN] + 5] = 0 } ->
110   L14_movl { memory[MAIN][esp[MAIN] + 4] = 0 } ->
111   L14_movl { eax[MAIN] = memory[MAIN][ebp[MAIN] - 3] } ->
112   L14_movl { memory[MAIN][esp[MAIN] + 3] = eax[MAIN] } ->
113   L14_movl { memory[MAIN][esp[MAIN] + 2] = filosofo_Proc } ->
114   L14_movl { memory[MAIN][esp[MAIN] + 1] = 0 } ->
115   L14_movl { memory[MAIN][esp[MAIN]] = 0 } ->
116   _CreateThread(filosofo_Proc, MAIN)
117   ;L14_movl { memory[HEAP][_threads + ebx[MAIN]] = eax[MAIN] } ->
118   L14_leal { eax[MAIN] = ebp[MAIN] - 2 } ->
119   L14_incl { memory[MAIN][eax[MAIN]] ++ } ->
120   L14()
121  };
122
123 L15() =
124  L15_movl { memory[MAIN][esp[MAIN] + 3] = -1 } ->
125  L15_movl { memory[MAIN][esp[MAIN] + 2] = 1 } ->
126  L15_movl { memory[MAIN][esp[MAIN] + 1] = _threads } ->
127  L15_movl { memory[MAIN][esp[MAIN]] = 3 } ->
128  _WaitForMultipleObjects(_threads, 3)
```

```
129  ;L15_movl { ebx[MAIN] = memory[MAIN][ebp[MAIN] -1] } ->
130  L15_leave { esp[MAIN] = ebp[MAIN]; ebp[MAIN] = memory[MAIN][esp[MAIN]]; esp[MAIN]-- } ->
131  L15_ret {  esp[MAIN] = esp[MAIN] + 1 } ->
132  End ->
133  Skip;
134
135 _filosofo(id) =
136  start.filosofo_Proc.id ->
137  _filoso_Body(id);
138  _ExitThread(id);
139
140 _filoso_Body(id) =
141  _filosofo_pushl.id { esp[id]--; memory[id][esp[id]] = ebp[id] } ->
142  _filosofo_movl.id { ebp[id] = esp[id] } ->
143  _filosofo_subl.id { esp[id] = esp[id] - 2 } ->
144  _filosofo_movl.id { eax[id] = memory[id][ebp[id] + 2] } ->
145  _filosofo_movl.id { memory[id][ebp[id] - 1] = eax[id] } ->
146  _filosofo_movl.id { eax[id] = memory[id][ebp[id] - 1] } ->
147  _filosofo_movl.id { eax[id] = memory[HEAP][eax[id]] } ->
148  _filosofo_movl.id { memory[id][esp[id]] = eax[id] } ->
149  call_pegarTalherEsq.id { esp[id]--; memory[id][esp[id]] = pc_dummy } ->
150  _pegarTalherEsq(id);
151  _filosofo_movl.id { eax[id] = memory[id][ebp[id] - 1] } ->
152  _filosofo_movl.id { eax[id] = memory[HEAP][eax[id]] } ->
153  _filosofo_movl.id { memory[id][esp[id]] = eax[id] } ->
154  call_pegarTalherDir.id { esp[id]--; memory[id][esp[id]] = pc_dummy } ->
155  _pegarTalherDir(id);
156  _filosofo_movl.id { eax[id] = memory[id][ebp[id] - 1] } ->
157  _filosofo_movl.id { eax[id] = memory[HEAP][eax[id]] } ->
158  _filosofo_movl.id { memory[id][esp[id]] = eax[id] } ->
159  call_comer.id { esp[id]--; memory[id][esp[id]] = pc_dummy } ->
160  _comer(id)
161  ;_filosofo_movl.id { eax[id] = memory[id][ebp[id] - 1] } ->
162  _filosofo_movl.id { eax[id] = memory[HEAP][eax[id]] } ->
163  _filosofo_movl.id { memory[id][esp[id]] = eax[id] } ->
164  call_devolverTalherEsq.id { esp[id]--; memory[id][esp[id]] = pc_dummy } ->
165  _devolverTalherEsq(id);
166  _filosofo_movl.id { eax[id] = memory[id][ebp[id] - 1] } ->
167  _filosofo_movl.id { eax[id] = memory[HEAP][eax[id]] } ->
168  _filosofo_movl.id { memory[id][esp[id]] = eax[id] } ->
169  call_devolverTalherDir.id { esp[id]--; memory[id][esp[id]] = pc_dummy } ->
170  _devolverTalherDir(id);
171  _filosofo_leave.id {
172   esp[id] = ebp[id];
173   ebp[id] = memory[id][esp[id]];
174   esp[id]++
175  } ->
176  _filosofo_ret.id { esp[id]++ } ->
177  Skip;
178
179 _pegarTalherEsq(id) =
180  _pegarTalherEsq_pushl.id { esp[id]--; memory[id][esp[id]] = ebp[id] }->
181  _pegarTalherEsq_movl.id { ebp[id] = esp[id] } ->
182  _pegarTalherEsq_subl.id { esp[id] = esp[id] - 6 } ->
```

```
183  _pegarTalherEsq_movl.id { eax[id] = memory[id][ebp[id] + 2] } ->
184  _pegarTalherEsq_movl.id { memory[id][ebp[id] - 1] = eax[id] } ->
185  _pegarTalherEsq_movl.id { memory[id][ebp[id] - 2] = 0 }->
186  L3(id);
187
188 L3(id) =
189  L3_cmpl.id { cmps[id] = 0 - memory[id][ebp[id] - 2] } ->
190  ifa( cmps[id] != 0 )
191  { L3_then.id -> L2(id) }
192  else
193  {
194   L3_else.id ->
195   L3_movl.id { memory[id][esp[id] + 1] = -1 } ->
196   L3_movl.id { eax[id] = _Mutex } ->
197   L3_movl.id { memory[id][esp[id]] = eax[id] } ->
198   LockMutex!id ->
199   L3_movl.id { eax[id] = memory[id][ebp[id] - 1] }->
200   L3_cmpl.id { cmps[id] = 0 - memory[HEAP][_garfos + eax[id]] } ->
201   ifa( cmps[id] != 0 )
202   { L3_then.id -> L5(id) }
203   else
204   {
205    L3_else.id ->
206    L3_movl.id { edx[id] = memory[id][ebp[id] - 1] } ->
207    L3_movl.id { eax[id] = memory[id][ebp[id] + 2] } ->
208    L3_incl.id { eax[id]++ } ->
209    L3_movl.id { memory[HEAP][_garfos + edx[id]] = eax[id] } ->
210    L3_movl.id { memory[id][ebp[id] - 2] = 1 } ->
211    L5(id)
212   }
213  };
214
215 L5(id) =
216  L5_movl.id { eax[id] = _Mutex }->
217  L5_movl.id { memory[id][esp[id]] = eax[id] } ->
218  ReleaseMutex!id ->
219  L3(id);
220
221 L2(id) =
222  L2_leave.id {
223   esp[id] = ebp[id];
224   ebp[id] = memory[id][esp[id]];
225   esp[id]++
226  } ->
227  L2_ret.id { esp[id]++ } ->
228  Skip;
229
230 _pegarTalherDir(id) =
231  _pegarTalherDir_pushl.id { esp[id]--; memory[id][esp[id]] = ebp[id] } ->
232  _pegarTalherDir_movl.id { ebp[id] = esp[id] } ->
233  _pegarTalherDir_subl.id { esp[id] = esp[id] - 6 } ->
234  _pegarTalherDir_movl.id { ecx[id] = memory[id][ebp[id] + 2] } ->
235  _pegarTalherDir_incl.id { ecx[id] ++ } ->
236  _MOD3_especial.id { eax[id] = ecx[id] - ((ecx[id]/3)*3) } ->
```

```
237   _pegarTalherDir_movl.id { memory[id][ebp[id] - 1] = eax[id] } ->
238   _pegarTalherDir_movl.id { memory[id][ebp[id] - 2] = 0 } ->
239   L7(id);
240
241 L7(id) =
242   L7_cmpl.id { cmps[id] = 0 - memory[id][ebp[id] - 2] } ->
243   ifa( cmps[id] != 0 )
244   { L7_then.id -> L6(id) }
245   else
246   {
247     L7_else.id ->
248     L7_movl.id { memory[id][esp[id] + 1] = -1 } ->
249     L7_movl.id { eax[id] = _Mutex } ->
250     L7_movl.id { memory[id][esp[id]] = eax[id] } ->
251     LockMutex!id ->
252     L7_movl.id { eax[id] = memory[id][ebp[id] - 1] } ->
253     L7_cmpl.id { cmps[id] = 0 - memory[HEAP][_garfos + eax[id]] } ->
254     ifa( cmps[id] != 0 )
255     { L7_then.id -> L9(id) }
256     else
257     {
258       L7_else.id ->
259       L7_movl.id { edx[id] = memory[id][ebp[id] - 1] } ->
260       L7_movl.id { eax[id] = memory[id][ebp[id] + 2] } ->
261       L7_incl.id { eax[id]++ } ->
262       L7_movl.id { memory[HEAP][_garfos + edx[id]] = eax[id] } ->
263       L7_movl.id { memory[id][ebp[id] - 2] = 1 } ->
264       L9(id)
265     }
266   };
267
268 L9(id) =
269   L9_movl.id { eax[id] = _Mutex }->
270   L9_movl.id { memory[id][esp[id]] = eax[id] } ->
271   ReleaseMutex!id ->
272   L7(id);
273
274 L6(id) =
275   L6_leave.id {
276     esp[id] = ebp[id];
277     ebp[id] = memory[id][esp[id]];
278     esp[id]++
279   } ->
280   L6_ret.id { esp[id]++ } ->
281   Skip;
282
283 _comer(id) =
284   _comer_push { esp[id]--; memory[id][esp[id]] = ebp[id] } ->
285   _comer_movl { ebp[id] = esp[id] } ->
286   _comer_popl { ebp[id] = memory[id][esp[id]]; esp[id]++ } ->
287   _comer_ret { esp[id]++ } ->
288   Skip;
289
290 _devolverTalherEsq(id) =
```

```
291  _devolverTalherEsq_pushl.id { esp[id]--; memory[id][esp[id]] = ebp[id] } ->
292  _devolverTalherEsq_movl.id { ebp[id] = esp[id] } ->
293  _devolverTalherEsq_subl.id { esp[id]--} ->
294  _devolverTalherEsq_movl.id { eax[id] = memory[id][ebp[id] + 2] } ->
295  _devolverTalherEsq_movl.id { memory[id][ebp[id] - 1] = eax[id] } ->
296  _devolverTalherEsq_movl.id { eax[id] = memory[id][ebp[id] - 1]  } ->
297  _devolverTalherEsq_movl.id { memory[HEAP][_garfos + eax[id]] = 0 } ->
298  _devolverTalherEsq_leave.id {
299   esp[id] = ebp[id];
300   ebp[id] = memory[id][esp[id]];
301   esp[id]++
302  } ->
303  _devolverTalherEsq_ret.id { esp[id]++ } ->
304  Skip;
305
306 _devolverTalherDir(id) =
307  _devolverTalherDir_pushl.id { esp[id]--; memory[id][esp[id]] = ebp[id] }->
308  _devolverTalherDir_movl.id { ebp[id] = esp[id] } ->
309  _devolverTalherDir_subl.id { esp[id]-- } ->
310  _devolverTalherDir_movl.id { ecx[id] = memory[id][ebp[id] + 2] } ->
311  _devolverTalherDir_incl.id { ecx[id] ++ } ->
312  _MOD3_especial.id { eax[id] = ecx[id] - ((ecx[id]/3)*3) } ->
313  _devolverTalherDir_movl.id { memory[id][ebp[id] - 1] = eax[id] } ->
314  _devolverTalherDir_movl.id { eax[id] = memory[id][ebp[id] - 1] } ->
315  _devolverTalherDir_movl.id { memory[HEAP][_garfos + eax[id]] = 0 } ->
316  _devolverTalherDir_leave.id {
317   esp[id] = ebp[id];
318   ebp[id] = memory[id][esp[id]];
319   esp[id]++
320  } ->
321  _devolverTalherDir_ret.id { esp[id]++ }->
322  Skip;
323
324 #assert _main() deadlockfree;
325 #assert _main() nonterminating;
326 #assert _main() |= (<> call_comer.1) && (<> call_comer.2) && (<> call_comer.3);
327 #define objetivo (memory[HEAP][_garfos] == 1 &&
328         memory[HEAP][_garfos + 1] == 2 && memory[HEAP][_garfos + 2] == 3);
329 #assert _main() |= !(<> objetivo);
```