

# $CSP_M$ E $CSP_\#$ : UMA ANÁLISE EXPERIMENTAL DE PROBLEMAS CLÁSSICOS DE CONCORRÊNCIA

Trabalho de Conclusão de Curso  
Engenharia da Computação

**Matheus Levi Pereira Torres**

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

Matheus Levi Pereira Torres

***CSP<sub>M</sub> E CSP#: UMA ANÁLISE  
EXPERIMENTAL DE PROBLEMAS  
CLÁSSICOS DE CONCORRÊNCIA***

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. de Carvalho

UNIVERSIDADE DE PERNAMBUCO  
ESCOLA POLITÉCNICA DE PERNAMBUCO  
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Recife - PE, Brasil

7 de dezembro de 2011

Declaro que revisei o Trabalho de Conclusão de Curso sob o título “*CSP<sub>M</sub> E CSP<sub>#</sub>: UMA ANÁLISE EXPERIMENTAL DE PROBLEMAS CLÁSSICOS DE CONCORRÊNCIA*”, de autoria de *Matheus Levi Pereira Torres*, e que estou de acordo com a entrega do mesmo.

Recife, \_\_\_\_ / \_\_\_\_\_ / \_\_\_\_

---

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

A Lindalva, Aurora, Miro e Julio

# *Agradecimentos*

Primeiramente, a Lindalva, minha avó materna, cuja história sempre me inspirou a superar desafios, e cujo conhecimento sempre me incentivou a expandir meus horizontes.

Aos meus pais, Laura e Julio, assim como meus irmãos Mirela e Eduardo, pelo apoio e incentivo a buscar minha formação, pela paciência, pelo carinho e pela dedicação.

Ao meu professor orientador Gustavo Carvalho, por sua excelente orientação e sua dedicação terem contribuído enormemente para a realização deste trabalho.

Aos colegas de faculdade Leandro, Pedro e Péricles, cuja convivência durante o curso tornou esta experiência ainda mais agradável, e cuja amizade atravessou as fronteiras da universidade.

Aos amigos Vinícius e Sairo, pela amizade de todas as horas, muito requisitada durante o período de desenvolvimento deste trabalho.

Por fim, aos amigos Poliana, Egon, Cássio, Gabriel e Raphael, pela dignidade que os tornaram pessoas essenciais em minha vida.

# *Resumo*

Durante o desenvolvimento de sistemas concorrentes, é comum encontrar fenômenos não visualizados em sistemas sequenciais, como *deadlock*, não-determinismo e *livelock*. Existem ferramentas que permitem analisar formalmente modelos mais abstratos da implementação com o intuito de antever futuros problemas de desenvolvimento. Essas ferramentas usam notações como CSP (*Communicating Sequential Processes*) e seus dois dialetos:  $\text{CSP}_M$  e  $\text{CSP}_\#$ . Apesar de algumas semelhanças, estas duas implementações diferem principalmente quanto ao modelo de comunicação adotado: a primeira só permite comunicação por troca de mensagens, enquanto que a segunda também permite por memória compartilhada. No entanto, é possível simular memória compartilhada em  $\text{CSP}_M$ . Neste trabalho, foram realizados experimentos a fim de determinar qual ferramenta, e seu respectivo dialeto, possui um melhor desempenho para avaliar propriedades clássicas de sistemas concorrentes em função do modelo de comunicação adotado pela especificação. Os resultados demonstraram que, embora seja difícil generalizar, a ferramenta PAT (*Process Analysis Toolkit*), e a implementação  $\text{CSP}_\#$ , obteve um desempenho melhor na maioria das situações avaliadas por este trabalho.

**Palavras-chave:** CSP,  $\text{CSP}_M$ ,  $\text{CSP}_\#$ , FDR, PAT.

# *Abstract*

During the development of concurrent systems, it is often found phenomena not seen in sequential systems, such as deadlock, non-determinism and livelock. There are tools which allow formal analysis of abstract models. This analysis aims to foresee future development issues. These tools use notations such as CSP (Communicating Sequential Processes) and its two dialects:  $\text{CSP}_M$  and  $\text{CSP}_\#$ . Despite some similarities, these implementations differ mainly regarding its communication model: the former only allows communication through message exchange, whereas the later also supports shared memory. However, it is possible to simulate shared memory using  $\text{CSP}_M$ . In this study, experiments were conducted to determine which tool, as well as its respective dialect, has a better performance to assert classical properties of concurrent systems, in accordance with the communication model adopted by the specification. Results have shown that although it is difficult to provide general claims, PAT (Process Analysis Toolkit), and its implementation  $\text{CSP}_\#$ , have had a better overall performance considering the scenarios evaluated by this work.

**Keywords:** CSP,  $\text{CSP}_M$ ,  $\text{CSP}_\#$ , FDR, PAT.

# *Sumário*

<b>Lista de Figuras</b>	p. xi
<b>Lista de Tabelas</b>	p. xii
<b>Lista de Abreviaturas e Siglas</b>	p. xiii
<b>1 Introdução</b>	p. 14
1.1 Qualificação do Problema . . . . .	p. 14
1.2 Objetivos . . . . .	p. 15
1.2.1 Objetivos Específicos . . . . .	p. 15
1.3 Resultados e Impactos Esperados . . . . .	p. 16
1.4 Estrutura da Monografia . . . . .	p. 16
<b>2 Referencial Teórico</b>	p. 17
2.1 Conceitos Chaves . . . . .	p. 17
2.1.1 CSP . . . . .	p. 17
2.1.2 $CSP_M$ . . . . .	p. 19
2.1.3 $CSP_{\#}$ . . . . .	p. 21
2.1.4 Análise Comparativa . . . . .	p. 25
2.2 Trabalhos Relacionados . . . . .	p. 26
<b>3 Método de Pesquisa</b>	p. 28
3.1 Objetivos do Experimento . . . . .	p. 28
3.2 Planejamento do Experimento . . . . .	p. 28



3.2.1	Hipóteses . . . . .	p. 29
3.2.2	Variáveis e Escalas . . . . .	p. 29
3.2.3	Hipóteses Nulas e Alternativas . . . . .	p. 30
3.2.4	Projeto Experimental . . . . .	p. 31
3.2.5	Instrumentos de Apoio . . . . .	p. 32
3.3	Análise Estatística . . . . .	p. 32
3.3.1	Teste Estatístico de Hipótese . . . . .	p. 32
3.3.2	Intervalo de Confiança . . . . .	p. 33
3.4	Ameaças à Validade . . . . .	p. 33
3.4.1	Ameaças à Validade Interna . . . . .	p. 33
3.4.2	Ameaças à Validade Externa . . . . .	p. 34
3.4.3	Ameaças à Validade de Conclusão . . . . .	p. 34
<b>4</b>	<b>Experimento e Resultados</b>	p. 35
4.1	Memória Simulada em $CSP_M$ . . . . .	p. 35
4.2	Descrição e Especificação de Problemas Clássicos de Concorrência . . .	p. 36
4.2.1	<i>Dining Philosophers</i> . . . . .	p. 36
4.2.1.1	Especificações . . . . .	p. 37
4.2.2	<i>Cigarette Smokers</i> . . . . .	p. 42
4.2.2.1	Especificações . . . . .	p. 42
4.3	Resultados dos Experimentos . . . . .	p. 47
4.3.1	Resultados dos Testes de Hipótese e Intervalos de Confiança . .	p. 48
4.3.2	Médias de Desempenho . . . . .	p. 50
4.3.3	Desempenho de FDR e PAT . . . . .	p. 51
<b>5</b>	<b>Considerações Finais</b>	p. 54
5.1	Trabalhos Futuros . . . . .	p. 55

<b>Referências</b>	p. 56
<b>Apêndice A – Experimento E1 (Resultados)</b>	p. 58
<b>Apêndice B – Experimento E2 (Resultados)</b>	p. 59
<b>Apêndice C – Experimento E3 (Resultados)</b>	p. 60
<b>Apêndice D – Experimento E4 (Resultados)</b>	p. 61

## *Lista de Figuras*

1	Tela Principal do FDR. . . . .	p. 20
2	Tela Inicial do ProBE. . . . .	p. 21
3	Execução Passo-a-passo de um Processo no ProBE. . . . .	p. 22
4	Tela Inicial do PAT. . . . .	p. 24
5	Análise de Execução de Processos e Geração de Grafos em PAT. . . .	p. 24
6	Verificação de Propriedades em PAT. . . . .	p. 25

# *Lista de Tabelas*

1	Comparação de Funcionalidades de $CSP_M$ /FDR e $CSP_{\#}$ /PAT . . . . .	p. 26
2	Resultados do Experimento 1 (E1) - Médias de Desempenho . . . . .	p. 51
3	Resultados do Experimento 2 (E2) - Médias de Desempenho . . . . .	p. 51
4	Resultados do Experimento 3 (E3) - Médias de Desempenho . . . . .	p. 51
5	Resultados do Experimento 4 (E4) - Médias de Desempenho . . . . .	p. 52
6	Relação entre Experimentos e Validação de Hipóteses . . . . .	p. 52

# *Lista de Abreviaturas e Siglas*

ASCII	<i>American Standard Code for Information Interchange</i>
CCS	<i>Calculus of Communicating Systems</i>
CSP	<i>Communicating Sequential Processes</i>
CSP <sub>M</sub>	<i>Machine-readable Communicating Sequential Processes</i>
CSP#	<i>Communicating Sequential Processes</i>
CWB	<i>Concurrency Workbench</i>
FDR	<i>Failures Divergences Refinement</i>
LTL	<i>Linear Temporal Logic</i>
NUS	<i>National University of Singapore</i>
PAT	<i>Process Analysis Toolkit</i>
ProBE	<i>Process Behaviour Explorer</i>
SPIN	<i>Simple Promela Interpreter</i>

# 1 *Introdução*

Presentes no dia a dia, sistemas concorrentes ou de tempo-real podem ser encontrados executando as mais diversas tarefas. Ainda assim, o desenvolvimento destes sistemas normalmente encontra dificuldades, uma vez que consistem de diversos componentes que podem executar paralelamente, aumentando sua complexidade em função de como esses componentes podem interagir. Este capítulo apresenta o problema avaliado, os objetivos e os resultados esperados deste trabalho.

## 1.1 Qualificação do Problema

Concorrência, por natureza, introduz fenômenos não averiguados em sistemas sequenciais, como *deadlock*, *livelock* e não-determinismo. *Deadlock* é a situação na qual dois ou mais processos concorrentes aguardam a finalização uns dos outros, e uma vez que nenhum finaliza, permanecem em estado de espera constante. Similarmente, *livelock* também ocorre entre dois ou mais processos concorrentes, mas estes continuam em execução, embora sem progresso. Já o não-determinismo ocorre quando há diferentes alternativas de fluxo para um sistema sem um método de escolha da alternativa a ser executada. Em sistemas concorrentes, a execução de fluxos alternativos num processo pode depender de outros processos dentro do sistema, mas se estes não funcionam como o previsto, a escolha de fluxo pode se tornar aleatória.

Tais fenômenos surgem não a partir de componentes individuais de um sistema, mas sim da forma como eles interagem entre si. Dessa forma, para projetar estes sistemas de forma eficiente, são necessárias maneiras de se analisar e controlar essas interações.

Estudos dedicados à especificação formal de sistemas desenvolveram então linguagens de programação que pudessem facilitar o entedimento do funcionamento de sistemas concorrentes. Nesse contexto, surgiu em 1985 a notação *Communicating Sequential Processes* (CSP) (HOARE, 1985), capaz de expressar um sistema concorrente complexo como uma

composição de processos e eventos.

A partir da pesquisa e evolução da notação CSP original foram feitas melhorias diversas, incluindo a criação de novos dialetos para a linguagem. Os dialetos (ou padrões) discutidos neste artigo são o  $CSP_M$ , publicado em 1998, sendo este uma versão de CSP passível de leitura de máquina (SCATTERGOOD, 1998); e o  $CSP_\#$ , publicado em 2008, que aproximou CSP das linguagens de programação tradicionais, incluindo diferentes funcionalidades e uma ferramenta própria (SUN; LIU; DONG, 2008). A principal ferramenta para analisar especificações em  $CSP_M$  é FDR (*Failures Divergences Refinement*) (FORMALSYSTEMS, 1992-2009), enquanto que PAT (*Process Analysis Toolkit*) analisa especificações  $CSP_\#$ .

Ambos os dialetos  $CSP_M$  e  $CSP_\#$  podem ser utilizados para especificar e analisar o comportamento de um sistema concorrente, porém, dadas as diferentes características, funcionalidades e limitações de cada um, a escolha do dialeto que melhor se adeque ao sistema a ser descrito é essencial.

Portanto, o problema de pesquisa deste trabalho é: considerando uma especificação de sistemas concorrentes que utiliza comunicação por troca de mensagens ou memória compartilhada, qual ferramenta (FDR ou PAT) apresenta um melhor desempenho para analisar propriedades básicas como ausência de *deadlock*, não determinismo e ausência de divergência.

## 1.2 Objetivos

O objetivo deste trabalho é comparar os dialetos de especificação formal  $CSP_M$  e  $CSP_\#$  da linguagem CSP, a partir de problemas clássicos de concorrência, com o intuito de determinar, experimentalmente, para quais paradigmas de comunicação - troca de mensagens ou memória compartilhada - o uso de  $CSP_M$  ou  $CSP_\#$  será mais eficiente na especificação e verificação de sistemas concorrentes.

### 1.2.1 Objetivos Específicos

- Selecionar e estudar problemas clássicos de concorrência;
- Especificar estes problemas em  $CSP_M$  e  $CSP_\#$ , utilizando tanto comunicação através de memória compartilhada quanto troca de mensagens;

- Realizar experimentos comparando o desempenho de PAT e FDR ao analisar as propriedades de ausência de *deadlock*, não-determinismo e ausência de divergência nos problemas clássicos especificados.

## 1.3 Resultados e Impactos Esperados

O resultado esperado é a corroboração da hipótese que, enquanto a notação  $CSP_M$  é mais adequada para especificação e validação de sistemas usando mensagens como mecanismo de comunicação entre processos, a notação  $CSP\#$  mostra resultados melhores para especificações de sistemas com memória compartilhada.

O impacto esperado é que, através deste estudo, seja possível escolher mais facilmente qual destes dialetos é mais apropriado para especificação e verificação de um sistema real em função do modelo de comunicação adotado por este sistema.

## 1.4 Estrutura da Monografia

Este trabalho está estruturado nos seguintes capítulos:

- Capítulo 1: introdução ao trabalho, qualificação do problema, objetivos e resultados esperados;
- Capítulo 2: revisão da bibliografia, incluindo descrições mais detalhadas sobre  $CSP_M$  e  $CSP\#$  e suas ferramentas, além de comparações de funcionalidades entre os dois dialetos.
- Capítulo 3: descrição detalhada dos experimentos realizados neste trabalho, incluindo hipóteses, hipóteses nulas e alternativas, além da análise estatística.
- Capítulo 4: resultados deste trabalho, descrevendo os problemas de concorrência escolhidos, detalhando cada uma das especificações produzidas, incluindo ainda resultados dos experimentos e inferências feitas a partir deles. Tabelas com todos os valores produzidos pelos experimentos encontram-se nos apêndices deste trabalho.
- Capítulo 5: conclusões sobre o trabalho e sugestões para trabalhos futuros.



## 2 *Referencial Teórico*

Este capítulo apresenta uma visão geral dos conceitos-chaves ao entendimento deste trabalho, como também uma análise de trabalhos relacionados.

### 2.1 Conceitos Chaves

Esta seção descreve alguns dos conceitos utilizados neste trabalho.

#### 2.1.1 CSP

CSP é uma linguagem formal para descrição de padrões de comportamento e interação em sistemas concorrentes, fortemente baseada em álgebra e processos matemáticos. Foi formulada inicialmente em 1985 por Hoare (HOARE, 1985), e posteriormente evoluída por outros pesquisadores ao longo dos anos.

Dentre os pesquisadores que evoluíram CSP, destaca-se Roscoe, cujo trabalho refinou a teoria original e proveu a versão mais moderna e conhecida da linguagem, em sua forma de processo algébrico (ROSCOE, 1998).

Tanto no meio acadêmico como na indústria de *software*, CSP e seus padrões encontraram utilidade como ferramentas de especificação, análise e verificação de sistemas concorrentes ou de tempo-real, provendo entendimento do comportamento e dos problemas particulares que podem surgir quando há concorrência entre processos.

O código abaixo é um exemplo de especificação em CSP.

Código 2.1: Exemplo de Especificação em CSP.

```

1 P = a → b → SKIP
2 Q = a → b → STOP
3 R(i) = c.i → R(i+1)
4 T = P □ Q
5 U = P □ Q
6 S = P ; Q
7 V = P ||| Q
8 X = P || Q
   {a}

```

No código 2.1, temos algumas das principais construções de CSP utilizadas neste trabalho. Estas são:

- **a** e **b** são eventos, ações particulares e atômicas que são executadas ou sofridas por um processo.
- **P**, **Q**, **R**, **T**, **U**, **S**, **V** e **X** são processos, um conjunto de eventos. A descrição de um processo sempre deve terminar com outro processo ou com os processos especiais SKIP ou STOP.
- **SKIP** é um processo especial que determina que um determinado fluxo de um processo finalizou e foi bem-sucedido.
- **STOP** é um processo especial que determina que um determinado fluxo de um processo finalizou de forma mal-sucedida.
- **R(i)** é um processo parametrizado, onde **i** é um parâmetro, cujo valor pode mudar a execução ou o fluxo do processo.
- O evento **c.i** é chamado evento composto, descrevendo a execução do evento *c* para os possíveis valores de *i*.
- Nos processos **P**, **Q** e **R(i)**, temos o operador  $\rightarrow$ , denominado prefixo. À sua esquerda do operador sempre está um evento que, quando acontece, o sistema passa a se comportar como o processo à direita do operador.
- No processo **T**, temos uma **escolha externa** entre os processos **P** e **Q**. Num fluxo qualquer de **T**, apenas **P** ou apenas **Q** podem ser executados, sendo a probabilidade de escolha a mesma para ambos. A escolha é dita externa, pois não depende de **T**.
- No processo **U**, temos uma **escolha interna** entre os processos **P** e **Q**. Num fluxo qualquer de **T**, apenas **P** ou apenas **Q** podem ser executados, sendo a probabilidade de escolha a mesma para ambos. A escolha é dita interna, pois **T** não-deterministicamente escolhe entre **P** ou **Q**.

- No processo S, temos dois processos em sequência. P executa primeiro e, se finalizado com sucesso, Q executa em seguida.
- No processo V, temos os processos P e Q em *interleaving*; os processos executam em paralelo, mas não sincronizam em nenhum evento, ou seja, não trocam mensagens.
- No processo X, temos os processos P e Q em paralelo, sincronizando num determinado evento *a*; ou seja, os dois processos precisam realizar o evento *a* simultaneamente.

### 2.1.2 CSP<sub>M</sub>

Com o progresso das pesquisas em torno de CSP, surgiu a necessidade da criação de ferramentas que dessem suporte a sua utilização, assim como facilitar as etapas de especificação, análise e verificação de sistemas. Uma vez que CSP é uma linguagem expressa com álgebras matemáticas, no entanto, seu uso em ferramentas computacionais se demonstrou pouco acessível.

Para solucionar este problema, CSP foi adaptada em um padrão (ou dialeto) que fosse passível de leitura por máquina, com o intuito de encorajar a criação de novas ferramentas para a linguagem. Esse padrão, intitulado CSP<sub>M</sub>, foi desenvolvido por Bryan Scattergood e publicado pela primeira vez em 1998 (SCATTERGOOD, 1998).

O código abaixo é um exemplo de especificação em CSP<sub>M</sub>.

Código 2.2: Exemplo de Especificação em CSP<sub>M</sub>

```

1 P = a -> b -> SKIP
2 Q = a -> b -> STOP
3 R(x) = a.x -> R(x+1)
4 T = P [ ] Q
5 U = P |~| Q
6 S = P ; Q
7 V = P ||| Q
8 X = P [| {a} ||] Q
9
10 assert T :[ deadlock free [F] ]
11 assert T :[ divergence free [FD] ]

```

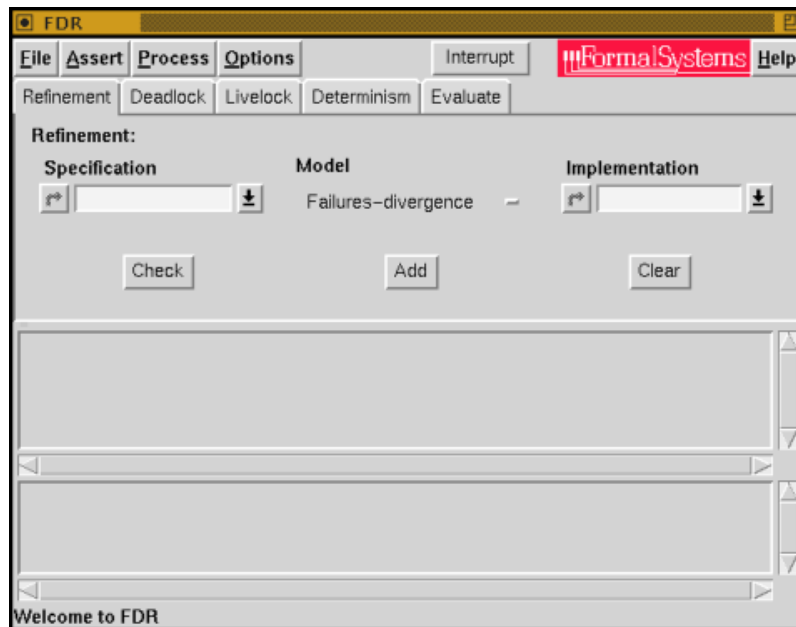
No exemplo de código 2.2, temos o mesmo código 2.1 adaptado para CSP<sub>M</sub>. Para ser passível de leitura por máquina, todos os operadores de CSP foram traduzidos para expressões em ASCII. Assim, dentre as principais diferenças, estão:

- O operador  $\rightarrow$ , que determina uma sequência de eventos, é substituído por  $->$ .
- O operador  $\square$ , que determina uma escolha externa, é substituído por  $[ ]$ .

- O operador  $\sqcap$ , que determina uma escolha interna, é substituído por  $\tilde{\sqcap}$ .

A partir de  $\text{CSP}_M$ , outras ferramentas foram desenvolvidas, tanto para o meio acadêmico como para a indústria de *software*. Neste trabalho, foram utilizadas as ferramentas FDR e ProBE, ambas desenvolvidas pela Formal Systems (FORMALSYSTEMS, 1986-2010).

Com o FDR, pode-se introduzir assertivas (*assertions*) no código, para verificação posterior da ferramenta. No exemplo, há duas afirmações: a primeira afirma que o processo T é livre de *deadlock* no modelo de falhas; a segunda afirma que o processo T é livre de divergência no modelo de falhas e divergência. Na ferramenta, essas afirmações aparecem listadas e podem ser testadas uma a uma, validando ou negando cada uma delas.



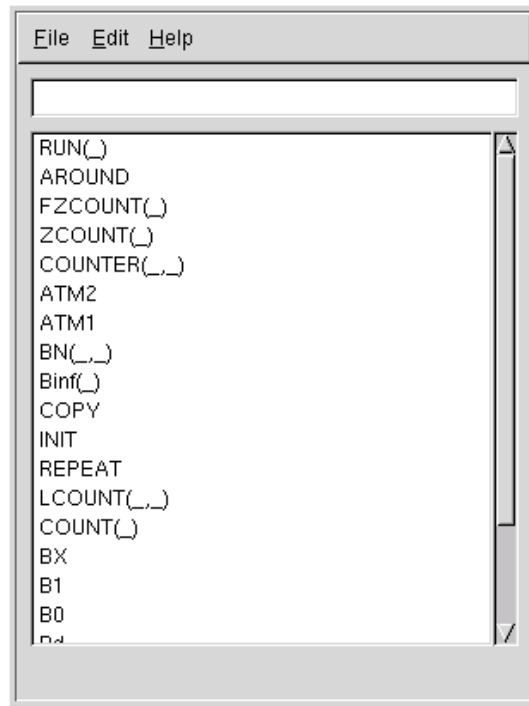
**Figura 1: Tela Principal do FDR.**

[Fonte: Manual *Online* do FDR2 FormalSystems (1992-2009)]

A Figura 1 é uma captura de tela do FDR em execução. Na parte superior, há abas com as opções de verificação disponíveis na ferramenta, dentre elas: refinamento, *deadlock*, *livelock* e determinismo. Após carregar um arquivo .csp externo com a especificação que se deseja analisar, no campo *Specification* pode-se informar o nome do processo a ser analisado.

No campo *Model*, escolhe-se o modelo da análise - na imagem acima, está selecionado o modelo de falhas e divergência. O retângulo vazio na parte central da imagem é o *Assertion List*, a lista de verificações a serem feitas, que é ser carregada diretamente do

arquivo de especificação. Por fim, o retângulo vazio na parte inferior da imagem é o *Process List*, onde são listados todos os processos definidos na especificação carregada pelo usuário.



**Figura 2: Tela Inicial do ProBE.**

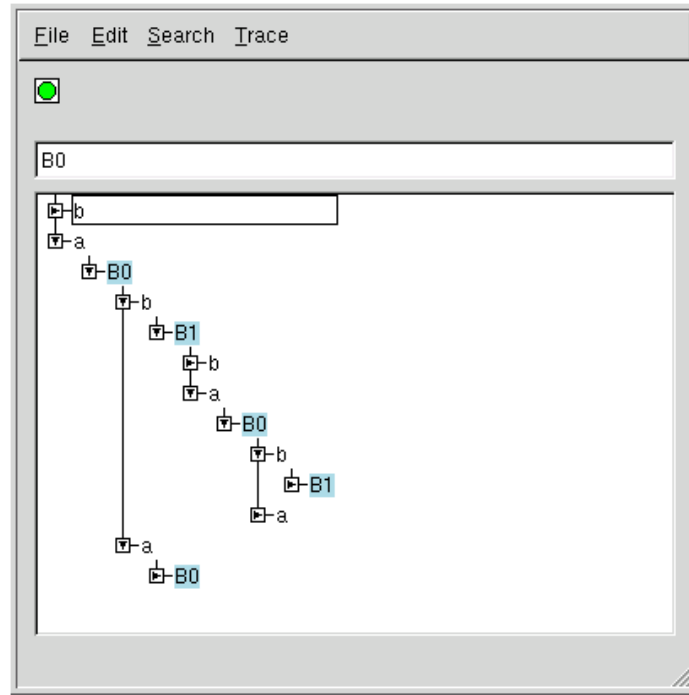
[Fonte: Manual *Online* do ProBE FormalSystems (2003)]

A Figura 2 é uma captura da tela inicial do ProBE. No menu *File* pode-se selecionar um arquivo externo .csp com a especificação que se deseja analisar. Em seguida são listados todos os processos descritos na especificação, como na figura. Com um clique duplo em qualquer um dos processos, abre-se uma nova janela para visualização de sua execução passo-a-passo.

A Figura 3 mostra a janela aberta após o clique duplo no processo B0 listado na Figura 3. São listados os possíveis eventos que podem acontecer no início do processo. Ao clicar em qualquer um dos eventos listados, o ProBE expande a lista com os eventos que podem ser executados em seguida, desenhando uma árvore de eventos e processos.

### 2.1.3 CSP#

Após a criação de  $CSP_M$  e suas principais ferramentas, os estudos em torno de CSP e técnicas de validação de modelos prosseguiram, evoluindo rapidamente. Naturalmente, as ferramentas utilizadas também se aprimoraram. Neste cenário, pesquisadores da NUS



**Figura 3: Execução Passo-a-passo de um Processo no ProBE.**

[Fonte: Manual *Online* do ProBE FormalSystems (2003)]

(*National University of Singapore*) revisaram o CSP original e, adicionando novas funcionalidades inspiradas em linguagens imperativas de programação de alto nível - em especial o C# da Microsoft - desenvolveram um novo dialeto, este chamado CSP#, e sua ferramenta de apoio, PAT (SUN; LIU; DONG, 2008).

As principais evoluções de CSP<sub>M</sub> para CSP# incluem a adição de variáveis de memória compartilhada, verificação de modelos baseada em lógica temporal (*LTL model check*), e o suporte à visualização e simulação da execução dos processos usando geração de grafos.

O código abaixo é um exemplo de especificação em CSP#.

**Código 2.3: Exemplo de Especificação em CSP#**

```

1 P() = a -> b -> Skip;
2 Q() = a -> b -> Stop;
3 R(x) = a.x -> R(x+1);
4 T() = P() [*] Q();
5 U() = P() <> Q();
6 S() = P() ; Q();
7 V() = P() ||| Q();
8 X() = P() || Q();
9
10 #assert T() deadlockfree;
11 #assert T() divergencefree;
```

No exemplo de código 2.3, temos o mesmo código 2.1 adaptada para CSP#. Para ser passível de leitura por máquina, todos os operadores de CSP foram traduzidos para

expressões em ASCII. Dentre as principais diferenças sintáticas, estão:

- Os processos especiais Skip e Stop não são escritos em caixa alta.
- Assim como em  $CSP_M$ , o operador  $\rightarrow$  que determina uma sequência de eventos é substituído por  $->$ .
- O operador  $\square$ , que determina uma escolha externa, é substituído por  $[*]$ .
- O operador  $\sqcap$ , que determina uma escolha interna, é substituído por  $<>$ .
- Tal como nas linguagens de programação tradicionais, cada expressão de código deve terminar com um  $;$  (ponto-e-vírgula).

$CSP\#$  ainda inclui um terceiro tipo de escolha, denominada *escolha geral*, cujo operador é  $[ ]$ . Embora os operadores sejam sintaticamente iguais, esta escolha não deve ser confundida com o operador de escolha externa de  $CSP_M$ .

Tal como em  $CSP_M$ , é possível inserir afirmações (*assertions*) num código em  $CSP\#$ . No exemplo, há duas afirmações: T é livre de *deadlock* e T é livre de divergência. Uma vez escritas, as afirmações são carregadas pela ferramenta PAT na janela *Verification*, como será mostrado a seguir.

A Figura 4 mostra a tela inicial da ferramenta PAT. Inicialmente temos um editor de texto com *syntax highlighting* (coloração de sintaxe) do dialeto  $CSP\#$  - uma vantagem em relação a  $CSP_M$ , que não possui editor próprio. Clicando no botão superior *Check Grammar* (ou apertando F5 no teclado), a ferramenta verifica se o código escrito está de acordo com a sintaxe do dialeto. Abaixo, em *Output Window*, são mostrados alertas ou erros do código fornecido.

Ao clicar em *Simulation*, uma nova janela se abre para visualização e análise da execução dos processos descritos, além da geração de grafos de estados e transições. Já em *Verification*, abre-se uma nova janela para execução das verificações descritas no código através do comando *assert*.

A Figura 5 mostra a simulação de processos em PAT. Pode-se visualizar uma simulação passo-a-passo através da função *Play Trace*, ou gerar um grafo com todos os estados e transições possíveis através da função *Generate Graph*. Ao lado dos botões, escolhe-se um processo para ser simulado ou ter seu grafo gerado, dentre os processos descritos no código da tela anterior.

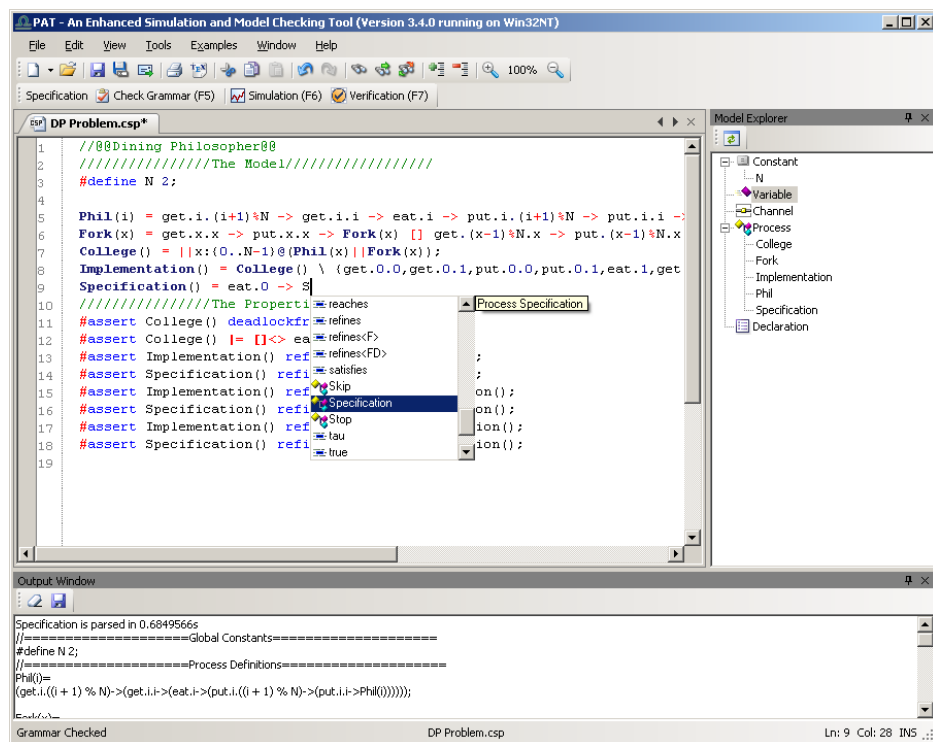


Figura 4: Tela Inicial do PAT.  
[Fonte: Manual *Online* do PAT Sun, Liu e Dong (2007)]

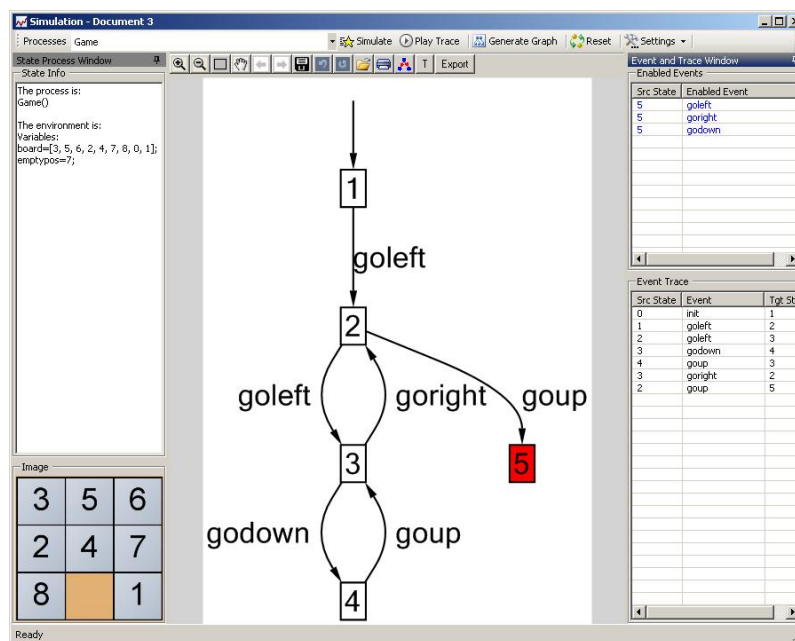
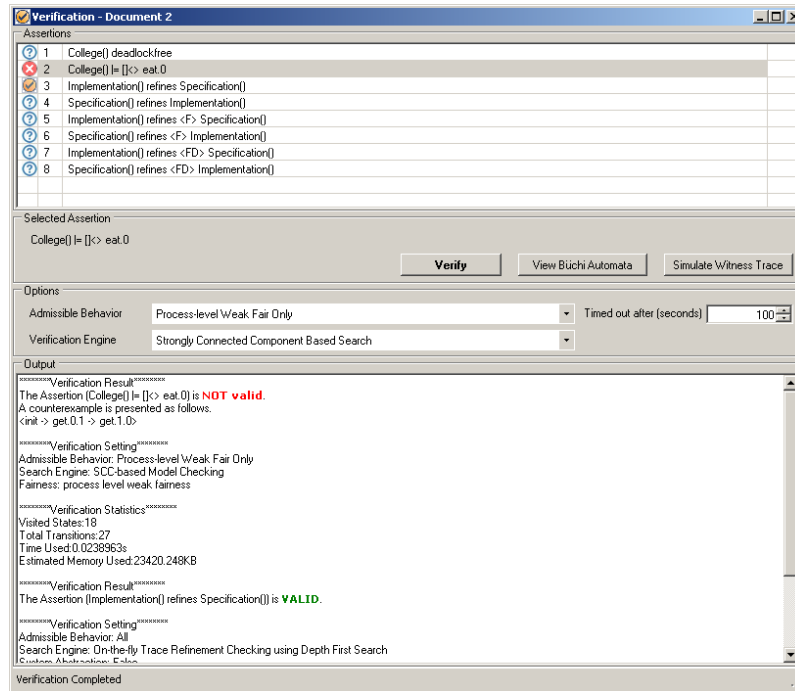


Figura 5: Análise de Execução de Processos e Geração de Grafos em PAT.  
[Fonte: Manual *Online* do PAT Sun, Liu e Dong (2007)]





**Figura 6: Verificação de Propriedades em PAT.**

[Fonte: Manual *Online* do PAT Sun, Liu e Dong (2007)]

A Figura 6 mostra a tela de verificação de propriedades na ferramenta PAT. São listados todos os *asserts* escritos no código da tela anterior, e com o botão *Verify*, todas as verificações são realizadas. Abaixo, em *Output*, a ferramenta informa se as verificações foram válidas ou não, provendo contra-exemplos para os casos negativos, além de dados estatísticos da verificação como número de estados visitados, total de transições, tempo consumido e quantidade estimada de memória do computador utilizada.

### 2.1.4 Análise Comparativa

Embora CSP# e PAT tenham sido desenvolvidos a partir de estudos mais recentes e incluído melhorias, CSP<sub>M</sub> e FDR ainda possuem funcionalidades importantes não encontradas em seus concorrentes. A principal delas é a sincronização múltipla de eventos através de comunicação utilizando mensagens, algo que CSP# permite apenas para um par de processos por vez.

A falta de memória compartilhada em CSP<sub>M</sub>, por outro lado, cria a necessidade de simular esta funcionalidade, criando um processo que atue como a memória do sistema. Neste trabalho, para o processo de simulação da memória foi utilizado um código adaptado do livro *The Theory and Practice of Concurrency* (ROSCOE, 1998), que será explicado com

mais detalhes no Capítulo 4.

Na Tabela 1 há uma comparação entre funcionalidades encontradas no dialeto  $\text{CSP}_M$  e sua ferramenta FDR, e no dialeto  $\text{CSP}_\#$  e sua ferramenta PAT.

**Tabela 1: Comparação de Funcionalidades de  $\text{CSP}_M$ /FDR e  $\text{CSP}_\#$ /PAT**  
[Fonte: elaboração própria]

	$\text{CSP}_M$ /FDR	$\text{CSP}_\#$ /PAT
Sincronização múltipla de eventos	Sim	Sim
Sincronização múltipla de eventos (com canais)	Sim	Não
Variáveis de memória compartilhada	Não	Sim
Análise de Alcançabilidade ( <i>Reachability</i> )	Não	Sim
Verificação baseada em lógica temporal	Não	Sim

## 2.2 Trabalhos Relacionados

Por se tratar de uma ferramenta relativamente recente, ainda há poucos trabalhos comparando FDR e PAT. Em particular, os autores de PAT (SUN; LIU; DONG, 2008) já analisaram algumas das diferenças entre seu dialeto e o  $\text{CSP}_M$ , bem como também testaram a eficiência das respectivas ferramentas. Neste artigo, experimentos foram realizados com os seguintes problemas de concorrência: *Dining Philosophers* (também utilizado neste trabalho), *Readers/Writers* e *Milner's Cyclic Scheduler* (SUN; LIU; DONG, 2008).

Para cada problema, os autores testaram verificações para diferentes valores de uma constante  $N$  que, em cada problema, tem significados diferentes:

- *Dining Philosophers*: número de filósofos.
- *Readers/Writers*: tamanho da memória (*buffer*).
- *Milner's Cyclic Scheduler*: número de tarefas agendadas (*scheduled tasks*).

Os resultados comparando PAT e FDR foram considerados satisfatórios pelos autores (SUN; LIU; DONG, 2008). Em *Dining Philosophers*, FDR teve um desempenho melhor em todos os casos, o que gerou uma melhoria da ferramenta PAT, especificamente para a análise deste problema. Em relação aos outros dois problemas, no entanto, PAT teve um desempenho melhor, incluindo casos em que FDR foi incapaz de executar a verificação em tempo hábil. Contudo, o artigo não deixa claro que modelo de comunicação foi adotado na implementação dos problemas em  $\text{CSP}_\#$ .

Em um outro artigo publicado, também com participação de autores de PAT, foi testada a verificação de linearizabilidade (*linearizability*) para diferentes tipos de estruturas de dados - *registers*, pilhas, filas, *mailboxes* e *SNZI* (LIU et al., 2009). Neste, os resultados foram novamente favoráveis a PAT, mas os experimentos incluíram apenas sistemas sem variáveis. Os autores consideraram que usar memória simulada em FDR tornaria os processos ainda mais lentos, portanto testes em  $CSP_M$  com memória não foram realizados.

Buscando uma abordagem diferente, este trabalho compara dois problemas de concorrência (*Dinning Philosophers* e *Cigarette Smokers*) utilizando dois paradigmas de comunicação entre processos: através de troca de mensagens e utilizando memória compartilhada. Com isso, pretende-se avaliar não apenas o desempenho das ferramentas em si, mas se um paradigma se demonstra mais adequado a uma determinada ferramenta.

## 3 *Método de Pesquisa*

Este trabalho consiste, na sua essência, no planejamento e execução de um experimento controlado com o intuito de responder o problema de pesquisa proposto. Portanto, este capítulo descreve, de acordo com as recomendações feitas por Wohlin (WOHLIN, 2000) e Juristo (JURISTO; MORENO, 2001), o planejamento do experimento controlado realizado por este trabalho.

### 3.1 Objetivos do Experimento

O objetivo do experimento é comparar os padrões de especificação formal  $CSP_{\#}$  e  $CSP_M$  e suas respectivas ferramentas, utilizando-se de problemas clássicos de concorrência, com o propósito de determinar experimentalmente qual destes padrões apresenta melhor eficiência nos modelos de comunicação baseados em troca de mensagens e memória compartilhada. Serão respondidas as seguintes perguntas:

- Questão 1 (Q1): Qual padrão apresenta melhor desempenho na análise de propriedades básicas, como ausência de *deadlock*, não-determinismo, ausência de divergência, no paradigma de troca de mensagens?
- Questão 2 (Q2): Qual padrão apresenta melhor desempenho na análise de propriedades básicas, como ausência de *deadlock*, não-determinismo, ausência de divergência, no paradigma de memória compartilhada?

### 3.2 Planejamento do Experimento

Esta seção descreve como o experimento foi planejado, desde a formulação das hipóteses até os instrumentos de apoio.

### 3.2.1 Hipóteses

Associadas às questões de pesquisa, assumem-se as seguintes hipóteses, a partir das quais elaborou-se o planejamento do estudo:

- Hipótese 1 (H1): a ferramenta PAT apresenta melhores resultados no paradigma de memória compartilhada, por dar suporte diretamente a esta propriedade em sua implementação.
- Hipótese 2 (H2): a ferramenta FDR apresenta melhores resultados no paradigma de troca de mensagens.

### 3.2.2 Variáveis e Escalas

- Fator 1 (F1): o padrão de CSP escolhido para descrever os problemas.
  - Alternativa 1 (F1-A1):  $CSP_M$
  - Alternativa 2 (F1-A2):  $CSP_\#$
  - Cada problema clássico será descrito em ambos os padrões, tanto para o paradigma de troca de mensagens, quanto para o paradigma de memória compartilhada.
    - \* Valores possíveis: descrição em  $CSP_M$  ou descrição em  $CSP_\#$ .
- Parâmetro 1 (P1): computador usado para os testes.
  - A execução das ferramentas FDR e PAT será feita usando um computador com a seguinte configuração:
    - \* Processador Core i3
    - \* Memória RAM 4GB
    - \* Sistema Operacional Windows 7 em Modo de Segurança (PAT 3.3.1)
    - \* Sistema Operacional Linux - Ubuntu 11.04 (FDR 2.91)

Uma vez que FDR e PAT não possuem versões para um mesmo sistema operacional, foram utilizados sistemas diferentes para os experimentos. No entanto, isso não caracteriza um fator, pois tal diferenciação não é aplicada à todos os experimentos, sendo uma limitação das ferramentas utilizadas.

- Parâmetro 2 (P2): número de filósofos.

- Para os testes envolvendo o problema de concorrência *Dining Philosophers*, fixou-se a constante N para o número de filósofos.
  - \* Único valor possível: 5.
- Parâmetro 3 (P3): o problema analisado.
  - Será considerado ou o problema dos filósofos (*Dining Philosophers*) ou o problema dos fumantes (*Cigarette Smokers*).
- Parâmetro 4 (P4): modelo de comunicação.
  - Será considerado ou o modelo de comunicação através de troca de mensagens ou por memória compartilhada.
- Variável de resposta 1 (R1): tempo (em segundos).
  - Será medida a partir da execução das ferramentas designadas a cada linguagem (FDR para  $CSP_M$ , PAT para  $CSP_\#$ ).

### 3.2.3 Hipóteses Nulas e Alternativas

- Hipótese Nula ( $H_{0-1}$ ):  $\mu_{R1,F1-A1} - \mu_{R1,F1-A2} = 0$ 
  - De acordo com esta hipótese nula, o valor médio de R1, quando escolhida a alternativa F1-A1, será similar ao valor médio de R1, quando escolhida a alternativa F1-A2. Ou seja, não há diferença entre o uso de FDR e PAT.
- Hipótese Alternativa 1 ( $H_{1-1}$ ):  $\mu_{R1,F1-A1} - \mu_{R1,F1-A2} < 0$ 
  - De acordo com esta hipótese alternativa, o valor médio de R1, quando escolhida a alternativa F1-A1, será menor que o valor médio de R1, quando escolhida a alternativa F1-A2. Ou seja, a ferramenta FDR se mostra mais rápida do que a ferramenta PAT.
- Hipótese Alternativa 2 ( $H_{1-2}$ ):  $\mu_{R1,F1-A1} - \mu_{R1,F1-A2} > 0$ 
  - De acordo com esta hipótese nula, o valor médio de R1, quando escolhida a alternativa F1-A1, será maior que o valor médio de R1, quando escolhida a alternativa F1-A2. Ou seja, a ferramenta PAT se mostra mais rápida que a ferramenta FDR.

### 3.2.4 Projeto Experimental

O experimento realizado por este trabalho é composto por vários sub-experimentos onde cada experimento fixa um valor para os parâmetros P3 e P4, antes descritos. Cada sub-experimento é realizado conforme o projeto experimental, um fator e dois tratamentos, descrito por Juristo e Moreno (2001). Para efeitos de tratamento estatístico, em cada sub-experimento será feita a coleta de R1 para 30 execuções.

- Experimento 1 (E1):
  - Modelo de Comunicação: Troca de Mensagens
  - Problema: *Cigarette Smokers*
  - Padrões de CSP:
    - \*  $CSP_M$  (F1-A1)
    - \*  $CSP_\#$  (F1-A2)
- Experimento 2 (E2):
  - Modelo de Comunicação: Memória Compartilhada
  - Problema: *Cigarette Smokers*
  - Padrões de CSP:
    - \*  $CSP_M$  (F1-A1)
    - \*  $CSP_\#$  (F1-A2)
- Experimento 3 (E3):
  - Modelo de Comunicação: Troca de Mensagens
  - Problema: *Dining Philosophers*
  - Padrões de CSP:
    - \*  $CSP_M$  (F1-A1)
    - \*  $CSP_\#$  (F1-A2)
- Experimento 4 (E4):
  - Modelo de Comunicação: Memória Compartilhada
  - Problema: *Dining Philosophers*
  - Padrões de CSP:

- \*  $CSP_M$  (F1-A1)
- \*  $CSP\#$  (F1-A2)

### 3.2.5 Instrumentos de Apoio

Como instrumentos de apoio a realização dos experimentos descritos acima, foram utilizados:

- Notepad++: ferramenta usada na descrição de processos em  $CSP_M$ , pois não há ferramentas de desenvolvimento dedicadas à linguagem.
- FDR: ferramenta usada na análise e verificação de processos no padrão  $CSP_M$ .
- ProBE: ferramenta usada na simulação passo-a-passo da execução de processos no padrão  $CSP_M$ .
- PAT: ferramenta usada na descrição, análise, simulação e verificação de processos no padrão  $CSP\#$ .
- GraphPad QuickCals: ferramenta *online* utilizada para realizar os cálculos da análise estatística do experimento (GRAPHPAD, 2002-2005).
- Microsoft Excel: elaboração de planilhas com os resultados.

## 3.3 Análise Estatística

Serão usadas duas técnicas para analisar a veracidade das hipóteses nulas e alternativas: um teste paramétrico e o cálculo do intervalo de confiança.

### 3.3.1 Teste Estatístico de Hipótese

Dado que este experimento será feito para um fator e duas alternativas, será feito uso do teste estatístico de hipótese *paired t-test*, normalmente aplicado a projetos experimentais deste tipo.

Para evitar ao máximo a ocorrência de qualquer erro estatístico, será assumido um nível de significância de 95%. Para o cálculo do *paired t-test*, será utilizada a seguinte fórmula:



$$t = \frac{\bar{d} - \delta}{S_d} \sqrt{N - 1}$$

Nesta fórmula,  $\bar{d}$  é a média das diferenças entre os resultados R1 quando aplicados F1-A1 e F2-A2, N é o número de elementos dos grupos analisados e  $S_d$  é o desvio padrão, calculado a partir da seguinte fórmula:

$$S_d = \sqrt{\frac{\sum (d - \bar{d})^2}{N - 1}}$$

### 3.3.2 Intervalo de Confiança

O intervalo de confiança complementa a análise feita anteriormente pelo teste de hipótese. Associado a um nível de confiança - que, neste trabalho, será de 95% - toma-se o intervalo de diferenças médias entre F1-A1 e F1-A2. Quanto menor o intervalo de confiança, maior a precisão do teste estatístico de hipótese. Caso o intervalo inclua um valor nulo (zero), entende-se que as amostras não são significativamente diferentes.

Abaixo, temos a fórmula de cálculo do intervalo de confiança:

$$IC = \bar{d} \pm t_{\alpha/2} \frac{S_d}{\sqrt{N}}$$

Na fórmula acima,  $\alpha$  vale 5% e é calculado a partir do nível de confiança  $(1 - \alpha)$ .

## 3.4 Ameaças à Validade

Apesar do cuidado com o planejamento do experimento, a sua validade está sujeita a algumas ameaças que serão aqui descritas.

### 3.4.1 Ameaças à Validade Interna

- **Experiência do autor:** dado que o autor do trabalho estudou os padrões CSP# e CSP<sub>M</sub> para a realização deste, a pouca experiência com as linguagens pode ter influenciado a qualidade das descrições dos problemas de concorrência escolhidos. Contudo, para minimizar esta ameaça, todas as especificações foram revisadas pelo orientador deste trabalho.

### 3.4.2 Ameaças à Validade Externa

- Complexidade dos problemas abordados: os problemas clássicos selecionados para este trabalho são conceitualmente simples, e, desse modo, geraram especificações também simples. Assim, os resultados obtidos através dos experimentos podem diferir bastante de problemas maiores e mais complexos.

### 3.4.3 Ameaças à Validade de Conclusão

- Baixo poder estatístico: em função do tempo curto para a realização deste trabalho, não foi possível realizar uma análise mais profunda das características da distribuição estatística dos dados coletados. Isto deveria ser feito para assegurar o uso do *paired t-test*. Contudo, para minimizar esta ameaça, todos os dados coletados encontra-se disponíveis no Apêndice A para que esta análise possa ser feita posteriormente.
- Confiabilidade das medidas: os testes foram realizados em um computador comum, nos sistemas operacionais Windows e Linux. É possível que, durante as realizações dos testes, programas ou serviços dos sistemas operacionais em execução possam ter consumido memória ou processamento do computador, prejudicando os testes e a coleta de dados do experimento.

## 4 *Experimento e Resultados*

Inicialmente, este capítulo irá apresentar o modelo utilizado para simular memória compartilhada em  $CSP_M$ . Em seguida, serão descritos os problemas selecionados, assim como os resultados dos experimentos realizados.

### 4.1 Memória Simulada em $CSP_M$

Uma vez que  $CSP_M$  não suporta variáveis de memória compartilhada nativamente, faz-se necessário simular a funcionalidade de memória usando processos e eventos.

O código utilizado neste trabalho foi adaptado do livro *The Theory and Practice of Concurrency* (ROSCOE, 1998). São estabelecidos dois conjuntos: `MEMORY_SIZE`, contendo os índices dos valores na memória (tal como índices de um *array*); e `VALUE_RANGE`, contendo os possíveis valores que um espaço na memória pode assumir.

Os canais *get* e *set* leem ou escrevem valores na memória, respectivamente. Cada célula de memória é representado pelo processo `CELL`, que recebe dois parâmetros: seu índice na memória e seu valor atual. Ao sincronizar com um evento de leitura (*get*), o valor atual da célula permanece o mesmo, com o processo `CELL` reiniciando com os mesmos parâmetros; já ao sincronizar com um evento de escrita (*set*), o valor da célula é alterado, e `CELL` reinicia recebendo este novo valor. A memória é, portanto, um conjunto de células independentes. Isto é modelado através do *interleaving* de vários processos `CELL` inicializados com valor 0. O Código 4.1 apresenta esta especificação.

Código 4.1: Memória Simulada em CSP<sub>M</sub>

```

1 FALSE = 0
2 TRUE = 1
3
4 MEMORY_SIZE = {0 .. 1}
5 VALUE_RANGE = {FALSE .. TRUE}
6
7 channel get : MEMORY_SIZE.VALUE_RANGE
8 channel set : MEMORY_SIZE.VALUE_RANGE
9
10 CELL(index, value) = get!index!value -> CELL(index, value)
11                      []
12                      set!index?new_value -> CELL(index, new_value)
13
14 MEMORY = ||| cell_index : MEMORY_SIZE @ CELL(cell_index, 0)

```

## 4.2 Descrição e Especificação de Problemas Clássicos de Concorrência

Dois problemas clássicos de concorrência foram selecionados e especificados em CSP<sub>M</sub> e CSP<sub>#</sub>, para ambos os paradigmas de troca de mensagens e memória compartilhada. Dessa forma, cada problema gerou quatro especificações diferentes.

Para que as especificações num mesmo dialeto, mas em paradigmas diferentes, representassem igualmente o mesmo problema, foram feitas verificações de refinamento no modelo semântico de falhas e divergência (*failure-divergence refinement check*). Ou seja, se a especificação utilizando troca de mensagem e a utilizando memória refinam, em uma dada linguagem, uma a outra. Dessa forma, buscou-se garantir que ambas especificações geram o mesmo diagrama de estados possível.

No entanto, para o problema *Dining Philosophers*, não foi possível o refinamento de falhas e divergência entre as especificações em CSP<sub>M</sub>, somente no modelo de *traces*, como será explicado durante o detalhamento destas especificações.

### 4.2.1 *Dining Philosophers*

Formulado originalmente por Edsger Dijkstra em 1965 (CHANDY; MISRA, 1984), neste problema cinco filósofos sentam-se em uma mesa circular para comer macarrão. Entre cada par de filósofos adjacentes há um garfo, e para que um filósofo se alimente, ele deve utilizar tanto seu garfo da esquerda como seu garfo da direita.

Cada filósofo pode pegar um garfo lateral, se disponível, e depois devolvê-lo à mesa quando não mais necessário - ou seja, após se alimentar de uma determinada quantidade

de comida. Para cada garfo pego ou devolvido há uma ação individual; ou seja, o filósofo não pode pegar ou devolver mais de um garfo em uma única ação.

#### 4.2.1.1 Especificações

- $CSP_M$  com troca de mensagens

Definida uma constante  $N$  para o número de filósofos, são descritos os processos PHILOSOPHER e FORK, ambos recebendo um parâmetro. O processo PHILOSOPHER representa um filósofo (determinado pelo parâmetro  $x$  recebido) que, em ordem, tenta executar as seguintes ações: pegar o garfo à sua direita (*take*), pegar o garfo à sua esquerda (*take*), comer, devolver o garfo à direita (*put*) e devolver o garfo à esquerda (*put*). Em seguida o processo volta a se comportar como PHILOSOPHER.

O processo FORK representa um garfo (determinado pelo parâmetro  $y$  recebido) que, em ordem, sincroniza com as seguintes ações de um filósofo: ser pego pelo filósofo à esquerda (*take*), em seguida devolvido à mesa (*put*) e voltando a se comportar como FORK; ou ser pego pelo filósofo à direita (*take*), em seguida devolvido à mesa (*put*) e voltando a se comportar como FORK.

Os processos auxiliares PHILOSOPHERS e FORKS criam um número  $N$  de processos PHILOSOPHER e FORK em *interleaving*, respectivamente. Em seguida, dentro do processo COLLEGE, ambos são sincronizados nas ações *take* e *put*. O código 4.2 apresenta esta especificação.

Código 4.2: *Dining Philosophers* em  $CSP_M$  com troca de mensagens

```

1 N = 5
2
3 Philosophers = {0 .. N-1}
4 Forks = {0 .. N-1}
5
6 channel take : Philosophers.Forks
7 channel put : Philosophers.Forks
8 channel eat : Philosophers
9
10 PHILOSOPHER(x) = take!x!(x+1)%N -> take!x!x -> eat!x -> put!x!(x+1)%N -> put!x!x
11   -> PHILOSOPHER(x)
12
13 PHILOSOPHERS = ||| i : {0..N-1} @ PHILOSOPHER(i)
14
15 FORK(y) = take!y!y -> put!y!y -> FORK(y) || take!(y-1)%N!y -> put!(y-1)%N!y
16   -> FORK(y)
17
18 FORKS = ||| i : {0..N-1} @ FORK(i)
19
20 COLLEGE = PHILOSOPHERS || {take, put} || FORKS

```

- CSP# com troca de mensagens

Definida uma constante  $N$  para o número de filósofos, são descritos os processos Phil e Fork, ambos recebendo um parâmetro cada. Phil representa um filósofo (determinado pelo parâmetro  $i$  recebido) que, em ordem, tenta executar as seguintes ações: pegar o garfo à sua direita (*get*), pegar o garfo à sua esquerda (*get*), comer, devolver o garfo à direita (*put*) e devolver o garfo à esquerda (*put*). Em seguida o processo volta a se comportar como Phil.

O processo Fork representa um garfo (determinado pelo parâmetro  $x$  recebido) que, em ordem, sincroniza com as seguintes ações de um filósofo: ser pego pelo filósofo à esquerda (*get*), em seguida devolvido à mesa (*put*) e voltando a se comportar como Fork; ou ser pego pelo filósofo à direita (*get*), em seguida devolvido à mesa (*put*) e voltando a se comportar como Fork.

Dentro do processo College, são criados  $N$  processos-filósofos e  $N$  processos-garfos, que sincronizam nas ações *get* e *put*.

O código 4.3 pode ser encontrado no manual oficial *online* da ferramenta PAT (SUN; LIU; DONG, 2007).

Código 4.3: *Dining Philosophers* em CSP# com troca de mensagens

```

1 #define N 5;
2
3 Phil(i) = get.i.(i+1)%N -> get.i.i -> eat.i -> put.i.(i+1)%N -> put.i.i -> Phil(i);
4 Fork(x) = get.x.x -> put.x.x -> Fork(x) [*] get.(x-1)%N.x -> put.(x-1)%N.x
5         -> Fork(x);
6 College() = || x:{0..N-1}@ (Phil(x) || Fork(x));

```

- CSP<sub>M</sub> com memória compartilhada SIMULADA

Definiu-se uma constante  $N$  para o número de filósofos, e índices de memória de 0 a  $2*N-1$ . Do índice 0 ao  $N-1$ , cada célula na memória representa se um determinado garfo está na mesa, ou seja, disponível para que um filósofo o pegue e use.

Uma vez que em CSP<sub>M</sub> não há como definir eventos ou processos atômicos, encontrou-se um problema durante a verificação da disponibilidade de um garfo. Verificar através de uma leitura na memória se um dado garfo está disponível, antes da ação *take*, poderia causar que dois processos-filósofos pegassem o mesmo garfo, se o segundo filósofo verifica a disponibilidade imediatamente após o primeiro, mas antes que este execute a ação *take* e atribua a disponibilidade do garfo como *false*.

Em virtude disso, buscando simular atomicidade de processos, foi criado o processo-controlador ATOMIC\_TAKE. Somente através deste processos os filósofos podem

realizar um *take*. Ou seja, este processo encapsula requisições de *take*, através do canal *atomic\_take* que são enviadas pelos filósofos. Ao receber esta requisição, este processo verifica se o garfo requisitado está na mesa, se estiver, o mesmo marca este como não mais na mesa e informa através do canal *atomic\_take\_feedback* que o garfo em questão foi pego com sucesso. Como somente este processo é capaz de realizar o *take*, se um processo tentar realizar um *take* durante a mesma ação de outro filósofo, ele terá que esperar, pois o processo *ATOMIC\_TAKE* só estará disponível para atender a requisição após responder a do primeiro filósofo.

Contudo, os filósofos precisam de um *feedback* se o garfo conseguiu ser pego ou não. Pois caso não tenha sido, ele continuará tentando pegá-lo, e caso tenha sido, ele agora tentará pegar o próximo garfo, ou comer, caso já tenha pego os dois garfos.

Diferentemente da especificação utilizando canais, o processo não fica bloqueado ao não conseguir pegar um garfo. Na prática ele fica enviando repetidamente requisições ao processo *ATOMIC\_TAKE*. Ou seja, não se tem um *deadlock*. Para gerar este *deadlock*, criou-se o processo auxiliar *CHECK\_DEADLOCK*.

O processo auxiliar *CHECK\_DEADLOCK* verifica se todas as posições de  $N$  a  $2*N-1$  são *TRUE* - ou seja, se os  $N$  filósofos estão tentando pegar um garfo, mas não estão conseguindo. Nesse caso, o sistema gera artificialmente um *deadlock*, pois nada mais pode ser escrito ou modificado. Para que o FDR interprete dessa maneira, força-se um *deadlock* com o evento *STOP*.

Ao analisar os estados gerados por esta especificação e executar o teste de refinamento no modelo de falhas e divergência em relação à *Dining Philosophers* em  $CSP_M$  usando troca de mensagens, percebeu-se que as especificações não refinam uma a outra, como buscou-se assegurar para o experimento. Isso acontece porque, inicialmente, no modelo com troca de mensagens, um filósofo  $i$  pode escolher entre duas ações: *take.i.0* (garfo à direita) e *take.i.1* (garfo à esquerda). Logo, tem-se um estado que não realizou ainda nenhum evento, mas que pode realizar ambos os *takes*.

Porém, nesta especificação, o evento *take* é encapsulado dentro do *ATOMIC\_TAKE*. Como este só trata uma requisição por vez, não se terá mais um estado que nenhum evento foi feito, mas que ambos os *takes* podem ser feitos. Em função disso, as especificações em  $CSP_M$  não refinam no modelo de falhas e falhas-divergência.

Ao fim da especificação, o processo *COLLEGE\_AUX* inicializa os  $N$  processos *PHIL\_MEM*. *COLLEGE\_MEM* engloba todos os processos, sincronizando *COL-*

LEGE\_AUX com o processo MEMORY nos eventos de leitura e escrita (*get* e *set*).

O código 4.4 apresenta esta especificação.

Código 4.4: *Dining Philosophers* em  $CSP_M$  com memória compartilhada simulada

```

1 N = 5
2
3 MEMORY_SIZE = {0 .. 2*N-1}
4 VALUE_RANGE = {FALSE .. TRUE}
5
6 Philosophers = {0 .. N-1}
7 Forks = {0 .. N-1}
8
9 channel atomic_take : Philosophers.Forks
10 channel atomic_take_feedback : Philosophers.VALUE_RANGE
11
12 FORKS_MEM_INIT(i) = if ( i < N ) then set!i!TRUE -> FORKS_MEM_INIT(i+1) else SKIP
13 BLOCKED_MEM_INIT(i) = if ( i < 2*N ) then set!i!FALSE
14                       -> BLOCKED_MEM_INIT(i+1) else SKIP
15
16 PHIL_MEM(i) =
17   PHIL_GETS_RIGHT(i)
18   ;
19   PHIL_GETS_LEFT(i)
20   ;
21   eat!i ->
22   put!i!(i+1)%N -> set!(i+1)%N!TRUE ->
23   put!i!i -> set!i!TRUE ->
24   PHIL_MEM(i)
25
26 ATOMIC_TAKE_MEMORY_CONTROLLER = ( [] fork_id : {0..N-1} @ ATOMIC_TAKE(fork_id) )
27   ;
28   ATOMIC_TAKE_MEMORY_CONTROLLER
29
30 ATOMIC_TAKE(fork_id) = atomic_take?i!fork_id -> get!fork_id?value ->
31                       if ( value == TRUE )
32                         then set!(i+N)!FALSE -> set!fork_id!FALSE
33                       -> take!i!fork_id -> atomic_take_feedback!i!TRUE -> SKIP
34                       else set!(i+N)!TRUE -> CHECK_DEADLOCK(N)
35
36   ;
37   atomic_take_feedback!i!FALSE -> SKIP
38
39 CHECK_DEADLOCK(pos) = if ( pos == 2*N ) then STOP
40                       else get!pos?value
41                       -> if ( value == TRUE ) then CHECK_DEADLOCK(pos+1)
42                       else SKIP
43
44 PHIL_GETS_RIGHT(i) =
45   atomic_take!i!(i+1)%N -> atomic_take_feedback!i?value ->
46   if ( value == TRUE ) then SKIP
47   else PHIL_GETS_RIGHT(i)
48
49 PHIL_GETS_LEFT(i) =
50   atomic_take!i!i -> atomic_take_feedback!i?value ->
51   if ( value == TRUE ) then SKIP
52   else PHIL_GETS_LEFT(i)
53
54 COLLEGE_AUX = ||| i : {0..N-1} @ PHIL_MEM(i)
55
56 COLLEGE_MEM = (MEMORY [] { |get, set| } [] (FORKS_MEM_INIT(0) ; BLOCKED_MEM_INIT(N)
57   ; (COLLEGE_AUX
58     [] { | atomic_take, atomic_take_feedback | } []
59     ATOMIC_TAKE_MEMORY_CONTROLLER ) ) )

```



- CSP# com memória compartilhada

Definida uma constante  $N$  para o número de filósofos, é criado um *array* chamado *fork\_available* de  $N$  posições, populado com booleanos de valor *true*. Estes booleanos representam se um determinado garfo está na mesa, ou seja, disponível para que um filósofo o pegue e use.

Os processos  $\text{Phil\_getRight}(i)$  e  $\text{Phil\_getLeft}(i)$  descrevem um filósofo  $i$  tentando pegar os garfos à sua direita e à sua esquerda, respectivamente. Ambos os processos são compostos por um evento atômico, onde uma guarda em espera ocupada verifica se o garfo que se deseja pegar está disponível. Em caso verdadeiro, o filósofo pega o garfo e a posição deste em *fork\_available* é atribuída para *false*, determinando que o garfo está indisponível.

O processo  $\text{Phil\_mem}(i)$ , onde o parâmetro  $i$  determina o filósofo em execução, é descrito como processos e eventos em sequência. Inicialmente ele tenta executar os processos  $\text{Phil\_getRight}(i)$  e  $\text{Phil\_getLeft}(i)$ . Se bem sucedido em ambos, o filósofo se alimenta (*eat*) e em seguida devolve os garfos da direita e da esquerda, atribuindo *true* à suas posições em *fork\_available*.

Por fim, o processo  $\text{College\_mem}()$  inicializa os  $N$  filósofos do problema, que não sincronizam em nenhum evento. O código 4.3 apresenta esta especificação.

Código 4.5: *Dining Philosophers* em CSP# com memória compartilhada

```

1 #define N 5;
2
3 // Forks
4 var fork_available[N] = [true(N)];
5
6 Phil_mem(i) =
7   Phil_getRight(i)
8   ;
9   Phil_getLeft(i)
10  ;
11  eat.i ->
12  put.i.(i+1)%N -> put_down_forks.i{fork_available[(i+1)%N]=true} ->
13  put.i.i -> put_down_forks.i{fork_available[i]=true} ->
14  Phil_mem(i);
15
16 Phil_getRight(i) =
17   atomic {
18     [ fork_available[(i+1)%N] == true ]
19     get.i.(i+1)%N -> set_fork_unavailable.i{fork_available[(i+1)%N]=false} -> Skip
20   };
21
22 Phil_getLeft(i) =
23   atomic{
24     [ fork_available[i] == true ]
25     get.i.i -> set_fork_unavailable.i{fork_available[i]=false} -> Skip
26   };
27
28 College_mem() = ||| x:{0..N-1}@ (Phil_mem(x));

```

### 4.2.2 *Cigarette Smokers*

Formulado originalmente por Suhas Patil em 1971 (PATIL, 1971), neste problema para se produzir e fumar um cigarro são necessários três ingredientes: tabaco, papel e fósforo. Há três fumantes em uma mesa, cada um com uma quantidade ilimitada de apenas um dos três ingredientes necessários - o primeiro com tabaco, o segundo com papel e o terceiro com fósforo. Além disso, um juiz escolhe, de forma arbitrária, dois fumantes para que estes coloquem na mesa os ingredientes que possuem, mas apenas a quantidade necessária para se fazer um único cigarro. Em seguida, o terceiro fumante é informado deste acontecimento, remove os ingredientes da mesa e os usa para produzir e fumar um cigarro. Uma vez que não há mais ingredientes na mesa, o juiz escolhe novamente dois fumantes para que coloquem seus ingredientes na mesa, reiniciando o processo.

#### 4.2.2.1 Especificações

- $CSP_M$  com troca de mensagens

São definidos quatro eventos, três representando ingredientes sendo compartilhados (*share\_tabacco*, *share\_paper* e *share\_fire*) e um quarto representando um dos fumantes produzindo seu cigarro e fumando-o (*smoke*).

Em seguida, três processos representam cada um dos três fumantes e seus ingredientes: TABACCO, PAPER e FIRE. Cada um destes processos tem apenas dois fluxos: compartilhar seu respectivo ingrediente e voltar ao estado inicial, ou fumar e voltar ao seu estado inicial.

Um quarto processo, JUDGE, representa o juiz. A partir de uma escolha interna, ele determina quais fumantes devem compartilhar seus ingredientes, sincronizando com os eventos *share\_tabacco*, *share\_paper* e *share\_fire*. O fumante restante então produz e fuma seu cigarro, e em seguida JUDGE volta o seu estado inicial.

O processo SMOKERS engloba todos os processos, sendo PAPER, TABACCO e FIRE em *interleaving* e os três sincronizando com JUDGE nos eventos *share* e *smoke*. O código 4.6 apresenta esta especificação.

Código 4.6: *Cigarette Smokers* em CSP<sub>M</sub> com troca de mensagens

```

1 smokers = {1 .. 3}
2 channel share_tabacco, share_paper, share_fire
3 channel smoke : smokers
4
5 TABACCO = share_tabacco -> TABACCO [] smoke.1 -> TABACCO
6 PAPER = share_paper -> PAPER [] smoke.2 -> PAPER
7 FIRE = share_fire -> FIRE [] smoke.3 -> FIRE
8
9 JUDGE = (share_paper -> SKIP ||| share_fire -> SKIP); smoke.1 -> JUDGE
10 |~| (share_tabacco -> SKIP ||| share_fire -> SKIP); smoke.2 -> JUDGE
11 |~| (share_tabacco -> SKIP ||| share_paper -> SKIP); smoke.3 -> JUDGE
12
13 SMOKERS = ((PAPER ||| TABACCO) ||| FIRE)
14 || {share_tabacco, share_paper, share_fire, smoke.1, smoke.2, smoke.3} || JUDGE

```

- CSP# com troca de mensagens

São definidos quatro eventos, três representando ingredientes sendo compartilhados (*share\_tabacco*, *share\_paper* e *share\_fire*) e um quarto representando um dos fumantes produzindo seu cigarro e fumando-o (*smoke*).

Em seguida, três processos representam cada um dos três fumantes e seus ingredientes: *Tabacco()*, *Paper()* e *Fire()*. Cada um destes processos tem apenas dois fluxos: compartilhar seu respectivo ingrediente e voltar ao estado inicial, ou fumar e voltar ao seu estado inicial.

Um quarto processo, *Judge()*, representa o juiz. A partir de uma escolha interna, ele determina quais fumantes devem compartilhar seus ingredientes, sincronizando com os eventos *share*. O fumante restante então produz e fuma seu cigarro, e em seguida *Judge()* volta o seu estado inicial.

O processo *Smokers()* engloba todos os processos, sendo *Paper()*, *Tabacco()* e *Fire()* em *interleaving* e os três sincronizando com *Judge()*. O código 4.7 apresenta esta especificação.

Código 4.7: *Cigarette Smokers* em CSP# com troca de mensagens

```

1 Tabacco() = share_tabacco -> Tabacco() [*] smoke.1 -> Tabacco();
2 Paper() = share_paper -> Paper() [*] smoke.2 -> Paper();
3 Fire() = share_fire -> Fire() [*] smoke.3 -> Fire();
4
5 Judge() = (share_paper -> Skip ||| share_fire -> Skip); smoke.1 -> Judge()
6 <& (share_tabacco -> Skip ||| share_fire -> Skip); smoke.2 -> Judge()
7 <& (share_tabacco -> Skip ||| share_paper -> Skip); smoke.3 -> Judge();
8
9 Smokers() = ( ( Paper() ||| Tabacco() ) ||| Fire() ) || Judge();

```

- CSP<sub>M</sub> com memória compartilhada SIMULADA

Para esta especificação, foram criadas nove posições na memória simulada, as quais podem assumir os valores FALSE (constante 0) ou TRUE (constante 1). As três

primeiras posições representam se um determinado fumante foi escolhido pelo juiz para compartilhar seu ingrediente (tabaco, papel e fósforo, respectivamente); as três seguintes representam se os ingredientes foram efetivamente compartilhados; e a última representa se um cigarro foi produzido e fumado por um dos três fumantes.

Cada fumante é representado como um conjunto de processos. Tomando como exemplo o fumante cujo ingrediente é o tabaco, tem-se o processo principal TABACCO\_MEM, como dois processos em *interleaving*: TABACCO\_SHARING e TABACCO\_SMOKING. O primeiro representa o fumante compartilhando seu ingrediente, caso seja escolhido pelo juiz; o segundo representa o fumante produzindo e fumando seu cigarro, caso os outros dois fumantes compartilhem seus ingredientes.

O processo TABACCO\_SHARING inicialmente sincroniza com qualquer outro processo que atribua TRUE para a posição 0 na memória, indicando o fumante foi escolhido para compartilhar seu ingrediente. Em seguida, uma vez compartilhado, o próprio processo atribui FALSE para a posição 0 e retorna ao estado inicial.

Já o processo TABACCO\_SMOKING é formado por um outro subconjunto de processos: TABACCO\_WAITING e TABACCO\_SKIP. O primeiro aguarda que outro processo atribua TRUE às posições de memória 4 e 5, informando que o ingredientes dos outros dois fumantes foram compartilhados. Em seguida, ao produzir seu cigarro e fumá-lo, ele atribui as posições 4 e 5 para FALSE, informando que os ingredientes foram utilizados, e atribui a posição 6 para TRUE, informando que produziu um cigarro e o fumou em seguida. Ao final, TABACCO\_WAITING retorna ao estado inicial.

A notação  $\text{PROCESSO\_1} \wedge \text{PROCESSO\_2}$  em  $\text{CSP}_M$  determina que o PROCESSO\_1 é executado inicialmente, mas é imediatamente interrompido caso o PROCESSO\_2 seja iniciado - ou seja, que algum evento inicial deste segundo aconteça. Dessa forma, TABACCO\_SKIP interrompe TABACCO\_WAITING caso algum outro fumante produza e fume o seu cigarro.

Para os fumantes com papel e fósforo, seus conjuntos de processos funcionam da mesma forma que o fumante do tabaco, alterando-se apenas as posições de memória onde necessário.

O processo JUDGE\_MEM inicia com uma escolha interna para atribuir TRUE em duas posições de memória das três iniciais, determinando quais fumantes deverão compartilhar seus ingredientes. Em seguida, o processo aguarda que um dos fumantes atribua TRUE à posição que determina que este fumou seu cigarro. Por fim, as

três últimas posições da memória são atribuídas FALSE, indicando que o fumante finalizou seu fumo, e o processo retorna ao estado inicial.

Por fim, o processo SMOKERS\_MEM engloba todos os processos, sendo TABACCO\_MEM, PAPER\_MEM, FIRE\_MEM e JUDGE\_MEM sincronizando com o processo MEMORY nos eventos de escrita e leitura de posições de memória (*get*, *set*). O código 4.8 apresenta esta especificação.

No código abaixo, foram utilizadas nove variáveis, enquanto no código em CSP# foram utilizadas sete variáveis. Foi verificado, no entanto, que duas variáveis a mais não prejudicou o desempenho desta especificação.

Código 4.8: *Cigarette Smokers* em CSP<sub>M</sub> com memória compartilhada simulada

```

1 FALSE = 0
2 TRUE = 1
3
4 MEMORY_SIZE = {0 .. 8}
5 VALUE_RANGE = {FALSE .. TRUE}
6
7 TABACCO_MEM = TABACCO_SHARING ||| TABACCO_SMOKING
8 TABACCO_SHARING = set .0.TRUE -> share_tabacco -> set !0!FALSE
9   -> set !3!TRUE -> TABACCO_SHARING
10 TABACCO_SMOKING = (TABACCO_WAITING /\ TABACCO_SKIP) ; TABACCO_SMOKING
11 TABACCO_WAITING = (set .4.TRUE -> SKIP ||| set .5.TRUE -> SKIP) ;
12   smoke.1 -> set !4!FALSE -> set !5!FALSE -> set !6!TRUE -> TABACCO_WAITING
13 TABACCO_SKIP = (smoke.2 -> SKIP [] smoke.3 -> SKIP) -> SKIP
14
15 PAPER_MEM = PAPER_SHARING ||| PAPER_SMOKING
16 PAPER_SHARING = set .1.TRUE -> share_paper -> set !1!FALSE
17   -> set !4!TRUE -> PAPER_SHARING
18 PAPER_SMOKING = (PAPER_WAITING /\ PAPER_SKIP) ; PAPER_SMOKING
19 PAPER_WAITING = (set .3.TRUE -> SKIP ||| set .5.TRUE -> SKIP) ;
20   smoke.2 -> set !3!FALSE -> set !5!FALSE -> set !7!TRUE -> PAPER_WAITING
21 PAPER_SKIP = (smoke.1 -> SKIP [] smoke.3 -> SKIP) -> SKIP
22
23 FIRE_MEM = FIRE_SHARING ||| FIRE_SMOKING
24 FIRE_SHARING = set .2.TRUE -> share_fire -> set !2!FALSE
25   -> set !5!TRUE -> FIRE_SHARING
26 FIRE_SMOKING = (FIRE_WAITING /\ FIRE_SKIP) ; FIRE_SMOKING
27 FIRE_WAITING = (set .3.TRUE -> SKIP ||| set .4.TRUE -> SKIP) ;
28   smoke.3 -> set !3!FALSE -> set !4!FALSE -> set !8!TRUE -> FIRE_WAITING
29 FIRE_SKIP = (smoke.1 -> SKIP [] smoke.2 -> SKIP) -> SKIP
30
31 JUDGE_MEM =
32 (
33   (set !1!TRUE -> SKIP ||| set !2!TRUE -> SKIP)
34   |~|
35   (set !0!TRUE -> SKIP ||| set !2!TRUE -> SKIP)
36   |~|
37   (set !0!TRUE -> SKIP ||| set !1!TRUE -> SKIP)
38 )
39 ;
40 (set .6.TRUE -> SKIP
41   []
42   set .7.TRUE -> SKIP
43   []
44   set .8.TRUE -> SKIP)
45 ; set !6!FALSE -> set !7!FALSE -> set !8!FALSE -> JUDGE_MEM
46
47 ALPHABET_TP = {set .3.TRUE, set .4.TRUE, set .5.TRUE, smoke.1, smoke.2, smoke.3}

```

```

48 | ALPHABET_TPF = {set .3.TRUE, set .4.TRUE, set .5.TRUE, smoke.1, smoke.2, smoke.3}
49 | ALPHABET_TPFJ = {set .0.TRUE, set .1.TRUE, set .2.TRUE,
50 |                 set .6.TRUE, set .7.TRUE, set .8.TRUE}
51 |
52 | SMOKERS_MEM = ( MEMORY [| {|get, set|}] [|
53 |   ( ( ( PAPER_MEM [| ALPHABET_TP [| TABACCO_MEM )
54 |     [| ALPHABET_TPF [| FIRE_MEM ) [| ALPHABET_TPFJ [| JUDGE_MEM ) )

```

- CSP# com memória compartilhada

Para este problema, foram criadas sete variáveis: três representam se um determinado fumante foi escolhido pelo juiz para compartilhar seu ingrediente (*tabacco\_chosen*, *paper\_chosen* e *fire\_chosen*); três representam se os ingredientes foram efetivamente compartilhados (*tabacco\_shared*, *paper\_shared* e *fire\_shared*); e a última representa se um cigarro foi produzido e fumado (*smoked*).

Cada processo representando um fumante (*Tabacco\_mem()*, *Paper\_mem()* e *Fire\_mem()*) consiste em uma escolha externa com duas guardas de espera ocupada. No primeiro fluxo, a guarda verifica se o fumante foi escolhido para compartilhar seu ingrediente; se sim, ele o compartilha atribuindo verdadeiro ao *shared* de seu ingrediente, e volta ao seu estado inicial após atribuir falso ao seu *chosen*.

No segundo fluxo, a guarda verifica se os outros dois ingredientes necessários ao fumante foram compartilhados; se sim, ele produz e fuma seu cigarro (atribuindo verdadeiro à *smoke*), em seguida alertando que não há mais ingredientes na mesa (atribuindo falso aos *shared* destes), e volta ao seu estado inicial.

Um quarto processo, *Judge\_mem()*, representa o juiz. A partir de uma escolha interna, ele determina quais fumantes devem compartilhar seus ingredientes, atribuindo verdadeiro às variáveis *chosen*. *Judge\_mem()* então aguarda que o fumante restante então produza e fume seu cigarro, com uma guarda de espera ocupada verificando se *smoked* é verdadeiro. Ao confirmar que alguém efetivamente fumou, *Judge\_mem()* atribui falso à *smoked* e retorna ao seu estado inicial.

Por fim, o processo *Smokers\_mem()* engloba os quatro processos em paralelo, sem sincronização. O código 4.9 apresenta esta especificação.

Código 4.9: *Cigarette Smokers* em CSP# com memória compartilhada

```

1 var tobacco_chosen = false;
2 var paper_chosen = false;
3 var fire_chosen = false;
4
5 var tobacco_shared = false;
6 var paper_shared = false;
7 var fire_shared = false;
8
9 var smoked = false;
10
11 Tabacco_mem() =
12 [tobacco_chosen] share_tobacco -> tobacco_unchosen{tobacco_chosen=false}
13   -> set_tobacco_shared{tobacco_shared=true} -> Tabacco_mem()
14 [*]
15 [paper_shared && fire_shared] smoke.1
16   -> set_paper_unshared{paper_shared=false}
17   -> set_fire_unshared{fire_shared=false}
18   -> set_tobacco_smoked{smoked=true} -> Tabacco_mem();
19
20 Paper_mem() =
21 [paper_chosen] share_paper -> paper_unchosen{paper_chosen=false}
22   -> set_paper_shared{paper_shared=true} -> Paper_mem()
23 [*]
24 [tobacco_shared && fire_shared] smoke.2
25   -> set_tobacco_unshared{tobacco_shared=false}
26   -> set_fire_unshared{fire_shared=false}
27   -> set_paper_smoked{smoked=true} -> Paper_mem();
28
29 Fire_mem() =
30 [fire_chosen] share_fire -> fire_unchosen{fire_chosen=false}
31   -> set_fire_shared{fire_shared=true} -> Fire_mem()
32 [*]
33 [tobacco_shared && paper_shared] smoke.3
34   -> set_tobacco_unshared{tobacco_shared=false}
35   -> set_paper_unshared{paper_shared=false}
36   -> set_fire_smoked{smoked=true} -> Fire_mem();
37
38 Judge_mem() =
39 (
40   choose_paper{paper_chosen=true} -> Skip
41   ||| choose_fire{fire_chosen=true} -> Skip
42   <>
43   choose_tobacco{tobacco_chosen=true} -> Skip
44   ||| choose_fire{fire_chosen=true} -> Skip
45   <>
46   choose_tobacco{tobacco_chosen=true} -> Skip
47   ||| choose_paper{paper_chosen=true} -> Skip
48 )
49 ;
50 [ smoked == true ] set_smoked_false{smoked=false} -> Judge_mem();
51
52 Smokers_mem() = ( ( Paper_mem() ||| Tabacco_mem() ) ||| Fire_mem() )
53   ||| Judge_mem();

```

## 4.3 Resultados dos Experimentos

Fazendo uma análise superficial dos códigos escritos para este trabalho, é possível perceber que as especificações em CSP<sub>M</sub>, usando memória compartilhada simulada, são mais complexas, o que refletiu numa maior dificuldade para escrever tais códigos. Uma vez

que as variáveis são acessadas através de números, a leitura e compreensão por terceiros também se torna mais difícil.

A falta de processos atômicos em  $CSP_M$  impediu ainda que a especificação de *Dining Philosophers* com memória compartilhada simulada refinasse aquela com troca de mensagens, para o refinamento de falhas e divergência, embora no modelo de *traces* as especificações são equivalentes.

Em  $CSP\#$ , além dos processos atômicos, o uso de guardas *booleanas* entre colchetes, designando uma guarda em espera ocupada, demonstrou-se bastante útil durante o desenvolvimento dos problemas. Essa funcionalidade garante que determinados eventos em um processo serão executados assim que o *boolean* interno torne-se *true*, evitando a necessidade de condicionais cíclicos como os utilizados em  $CSP_M$ . A sintaxe, mais próxima das linguagens de programação atuais também contribuiu para uma maior facilidade nas especificações em  $CSP\#$ .

### 4.3.1 Resultados dos Testes de Hipótese e Intervalos de Confiança

Para cada experimento, foram calculados os resultados do *paired t-test* (t) e o intervalo de confiança (IC) nas verificações de ausência de *deadlock*, não-determinismo e ausência de divergência:

- Experimento 1 (E1):

- Ausência de *Deadlock*:

- \*  $\mu_{R1,F1-A1} - \mu_{R1,F1-A2} = 0,008014189700$

- \*  $t = 4,3503$

- \*  $IC = [0,004246440492 \dots 0,011781938908]$

- \* Estatisticamente *significativo*

- Não-determinismo:

- \*  $\mu_{R1,F1-A1} - \mu_{R1,F1-A2} = 0,012704272233$

- \*  $t = 34,5229$

- \*  $IC = [0,011951637408 \dots 0,013456907059]$

- \* Estatisticamente *significativo*

- Ausência de divergência:

- \*  $\mu_{R1,F1-A1} - \mu_{R1,F1-A2} = 0,035212714400$



$$* t = 1,3889$$

$$* IC = [-0,016640892563 \dots 0,087066321363]$$

\* Estatisticamente *não-significativo*

- Experimento 2 (E2):

- Ausência de *Deadlock*:

$$* \mu_{R1,F1-A1} - \mu_{R1,F1-A2} = -12,421967758000$$

$$* t = 884,0069$$

$$* IC = [-12,450707110446 \dots -12,393228405554]$$

\* Estatisticamente *significativo*

- Não-determinismo:

$$* \mu_{R1,F1-A1} - \mu_{R1,F1-A2} = 0,032141417067$$

$$* t = 63,6446$$

$$* IC = [0,031108546900 \dots 0,033174287233]$$

\* Estatisticamente *significativo*

- Ausência de divergência:

$$* \mu_{R1,F1-A1} - \mu_{R1,F1-A2} = -36,884427101667$$

$$* t = 4499,0750$$

$$* IC = [-36,901194360089 \dots -36,867659843244]$$

\* Estatisticamente *significativo*

- Experimento 3 (E3):

- Ausência de *Deadlock*:

$$* \mu_{R1,F1-A1} - \mu_{R1,F1-A2} = 0,011162477500$$

$$* t = 10,4017$$

$$* IC = [0,008967656900 \dots 0,013357298100]$$

\* Estatisticamente *significativo*

- Não-determinismo:

$$* \mu_{R1,F1-A1} - \mu_{R1,F1-A2} = 0,006961258767$$

$$* t = 14,1767$$

$$* IC = [0,005956979159 \dots 0,007965538374]$$

\* Estatisticamente *significativo*

- Ausência de divergência:
  - \*  $\mu_{R1,F1-A1} - \mu_{R1,F1-A2} = -0,016679339867$
  - \*  $t = 5,4917$
  - \*  $IC = [-0,022891056912 \dots -0,010467622821]$
  - \* Estatisticamente *significativo*
- Experimento 4 (E4):
  - Ausência de *Deadlock*:
    - \*  $\mu_{R1,F1-A1} - \mu_{R1,F1-A2} = 0,698554399667$
    - \*  $t = 380.9007$
    - \*  $IC = [0,694803541380 \dots 0,702305257954]$
    - \* Estatisticamente *significativo*
  - Não-determinismo:
    - \*  $\mu_{R1,F1-A1} - \mu_{R1,F1-A2} = 3,028775415000$
    - \*  $t = 839,1113$
    - \*  $IC = [3,021393147628 \dots 3.036157682372]$
    - \* Estatisticamente *significativo*
  - Ausência de divergência:
    - \*  $\mu_{R1,F1-A1} - \mu_{R1,F1-A2} = 0,825542649000$
    - \*  $t = 376,9712$
    - \*  $IC = [0,821063726086 \dots 0,830021571914]$
    - \* Estatisticamente *significativo*

### 4.3.2 Médias de Desempenho

A seguir, temos a média de desempenho das ferramentas FDR e PAT para a verificação de ausência de *deadlock* e determinismo, para cada um dos problemas especificados:

- Experimento 1 (E1):

Os resultados do Experimento 1 estão descritos na Tabela 2. Podemos observar que PAT teve melhor desempenho em todos os casos.

**Tabela 2: Resultados do Experimento 1 (E1) - Médias de Desempenho**

[Fonte: Experimento 1 (E1)]

	<b>FDR</b>	<b>PAT</b>
<b>Ausência de <i>deadlock</i></b>	0,021092803033	<b>0,013078613333</b>
<b>Não-determinismo</b>	0,022504345567	<b>0,009800073333</b>

**Tabela 3: Resultados do Experimento 2 (E2) - Médias de Desempenho**

[Fonte: Experimento 2 (E2)]

	<b>FDR</b>	<b>PAT</b>
<b>Ausência de <i>deadlock</i></b>	<b>0,044523435333</b>	12,466491193333
<b>Não-determinismo</b>	0,042265860400	<b>0,010124443333</b>
<b>Ausência de divergência</b>	<b>0,042764381667</b>	36,927191483333

- Experimento 2 (E2):

Os resultados do Experimento 2 estão descritos na Tabela 3. Podemos observar que FDR teve melhor desempenho para ausência de *deadlock* e ausência de divergência.

- Experimento 3 (E3):

Os resultados do Experimento 3 estão descritos na Tabela 4. Podemos observar que FDR teve desempenho melhor apenas para ausência de divergência.

**Tabela 4: Resultados do Experimento 3 (E3) - Médias de Desempenho**

[Fonte: Experimento 3 (E3)]

	<b>FDR</b>	<b>PAT</b>
<b>Ausência de <i>deadlock</i></b>	0,029784687500	<b>0,018622210000</b>
<b>Não-determinismo</b>	0,041898952100	<b>0,034937693333</b>
<b>Ausência de divergência</b>	<b>0,032063223467</b>	0,048742563333

- Experimento 4 (E4):

Os resultados do Experimento 4 estão descritos na Tabela 5. Podemos observar que PAT teve desempenho melhor em todos os casos.

### 4.3.3 Desempenho de FDR e PAT

Após a análise estatística dos experimentos, pôde-se determinar que eles são estatisticamente significativos, pois possuem intervalos de confiança curtos que não incluem o valor nulo (zero). A única exceção é a verificação de ausência de divergência para o Experimento 1 (E1), cujo intervalo de confiança incluiu zero, tornando esta verificação estatisticamente não-significativa.

**Tabela 5: Resultados do Experimento 4 (E4) - Médias de Desempenho**

[Fonte: Experimento 4 (E4)]

	<b>FDR</b>	<b>PAT</b>
<b>Ausência de <i>deadlock</i></b>	0,717565839667	<b>0,019011440000</b>
<b>Não-determinismo</b>	3,091605871667	<b>0,062830456667</b>
<b>Ausência de divergência</b>	0,926621375667	<b>0,101078726667</b>

De acordo com as duas hipóteses iniciais (H1 e H2), PAT apresentariam resultados melhores (menores médias de desempenho) para o paradigma de memória compartilhada (ou seja, para os experimentos E2 e E4); e FDR apresentariam resultados melhores (menores médias de desempenho) para os paradigmas de troca de mensagens (ou seja, para os experimentos E1 e E3).

No entanto, como pode-se ver nas tabelas dos experimentos E1 e E4, PAT teve um desempenho melhor, até mesmo para troca de mensagens. No experimento E3, FDR executou em menos tempo apenas a verificação de ausência de divergência. De forma surpreendente, PAT teve desempenhos extremamente inferiores nas verificações de ausência de *deadlock* e ausência de divergência em E2. No caso específico de ausência de divergência desse experimento, PAT gerou grafos com mais de 45 mil estados e 2 milhões de transições, resultando na lentidão desta verificação. Percebe-se que um número muito maior de processos funcionando em paralelo não tornou lenta a execução da especificação em  $CSP_M$  no FDR.

A tabela 6 mostra a relação entre os experimentos realizados e a validação ou refutação das hipóteses nula e alternativas.

**Tabela 6: Relação entre Experimentos e Validação de Hipóteses**

[Fonte: Experimentos E1, E2, E3, E4]

<b>Experimento</b>	<b>Verificação</b>	<b><math>H_{0-1}</math></b>	<b><math>H_{1-1}</math></b>	<b><math>H_{1-2}</math></b>
<b>E1</b>	Ausência de <i>deadlock</i>	×	×	✓
	Não-determinismo	×	×	✓
	Ausência de divergência	✓	×	×
<b>E2</b>	Ausência de <i>deadlock</i>	×	✓	×
	Não-determinismo	×	×	✓
	Ausência de divergência	×	✓	×
<b>E3</b>	Ausência de <i>deadlock</i>	×	×	✓
	Não-determinismo	×	×	✓
	Ausência de divergência	×	✓	×
<b>E4</b>	Ausência de <i>deadlock</i>	×	×	✓
	Não-determinismo	×	×	✓
	Ausência de divergência	×	×	✓

O resultado do experimento E2 indica que o que determina um melhor desempenho entre  $CSP_M$  e  $CSP_{\#}$  pode ser mais sutil do que se pensa. Muito embora o código em

$CSP_M$  seja mais complexo que o código em  $CSP_{\#}$ , este último teve um tempo de execução extremamente maior, mesmo se tratando de um problema relativamente simples e com poucos processos. Em um sistema de características similares, mas com mais processos envolvidos, é possível que o tempo de execução de PAT aumente de tal forma que sua utilização seja inviável.

Ainda assim, como muitas vezes a complexidade de um sistema não é totalmente compreendida até antes de sua especificação formal ou até mesmo seu desenvolvimento, através dos experimentos E1, E3 e E4, pode-se garantir que PAT é uma escolha mais segura, tanto em relação a complexidade do código a ser escrito quanto ao desempenho da ferramenta. Como a Tabela 6 mostra, PAT foi melhor em 8 dos 12 cenários analisados (aproximadamente 66,67% dos casos).

## 5 *Considerações Finais*

Especificar e desenvolver sistemas para problemas concorrentes usualmente traz fenômenos não encontrados em sistemas sequenciais e com eles uma maior complexidade. Os principais fenômenos averiguados neste trabalho - *deadlock*, determinismo e *livelock* - são apenas alguns dos que podem surgir durante a execução de um sistema concorrente.

Visando uma melhor compreensão dos fenômenos, suas causas e formas de evitar ou minimizar suas ocorrências surgiu a linguagem CSP, mais tarde expandida e aprimorada, gerando dois dialetos:  $CSP_M$  e  $CSP\#$ .

Neste trabalho, as especificações foram bem-sucedidas em ambos os dialetos. No entanto, encontrou-se maior complexidade nas especificações em  $CSP_M$ , especialmente naquelas utilizando memória compartilhada simulada - uma vez que a linguagem não suporta memória compartilhada diretamente. A ausência de determinadas funcionalidades em  $CSP_M$  encontradas em  $CSP\#$ , como processos atômicos ou verificações em espera ocupada, ainda implicou em maiores dificuldades na etapa de especificação. Dessa forma, evidencia-se que códigos em  $CSP_M$  tendem a ser mais extensos.

Assumiram-se as hipóteses de que  $CSP\#$  apresentaria resultados melhores para as especificações usando memória compartilhada, enquanto  $CSP_M$  apresentaria resultados melhores para aquelas usando troca de mensagens. Os resultados dos experimentos, porém, demonstraram que PAT pode ter desempenho superior mesmo para troca de mensagens.

Chama a atenção ainda que, num dos experimentos, houve dois casos em que  $CSP\#$  teve desempenho bastante inferior, levantando questionamentos sobre quais fatores, senão a escolha do modelo de comunicação dos processos, determinam melhor eficiência para as ferramentas testadas.

Pode-se assumir que  $CSP\#$  e PAT são uma escolha mais segura por seus resultados melhores na maior parte dos testes e experimentos. Ainda assim, é ideal que seja feita uma análise prévia antes da especificação e verificação de um sistema, com o intuito de saber de que forma as funcionalidades (ou ausência de funcionalidades) de FDR e PAT

podem influir durante as etapas.

A especificação e verificação de um sistema concorrente antes de sua programação efetiva traz uma maior compreensão do que será desenvolvido e cria meios de se evitar *bugs* na execução destes sistemas. Ao final do processo, garante-se um *software* de maior qualidade e a prova de falhas comuns a sistemas concorrentes.

## 5.1 Trabalhos Futuros

Como trabalhos futuros complementares a este, destacam-se:

- Repetir os experimentos intercalando os testes de FDR e PAT, uma vez que, para este trabalho, os experimentos foram realizados de forma sequencial;
- Repetir os experimentos e verificações para mais problemas clássicos de concorrência;
- Para problemas que possuem uma ou mais constantes, como *Dining Philosophers* ou *Readers/Writers*, expandir os experimentos para diferentes valores das constantes;
- Repetir os experimentos incluindo outras linguagens de modelagem de sistemas, como CCS;
- Repetir os experimentos utilizando outras ferramentas de verificação de modelos, como CWB.

## *Referências*

- CHANDY, K. M.; MISRA, J. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, v. 6, p. 632–646, 1984.
- FORMALSYSTEMS. *Formal Systems Website*. 1986–2010. Disponível em: <http://www.fsel.com>.
- FORMALSYSTEMS. *FDR2 User Manual*. [S.l.], 1992–2009. Disponível em: <http://www.fsel.com/documentation/fdr2/html/fdr2manual.html>.
- FORMALSYSTEMS. *ProBE User Manual*. [S.l.], 2003. Disponível em: <http://www.fsel.com/documentation/probe/probe-doc-html/html/probe.html>.
- GRAPHPAD. *GraphPad QuickCalcs*. 2002–2005. Disponível em: <http://www.graphpad.com/quickcalcs/ttest1.cfm>.
- HOARE, C. A. R. Communicating sequential processes. *Communications of the ACM*, v. 21, p. 666–677, 1985.
- JURISTO, N.; MORENO, A. *Basics of software engineering experimentation*. Kluwer Academic Publishers, 2001. ISBN 9780792379904. Disponível em: <http://books.google.com/books?id=ovWfOeW653EC>.
- LIU, Y. et al. Model checking linearizability via refinement. In: *Proceedings of the 2nd World Congress on Formal Methods*. Berlin, Heidelberg: Springer-Verlag, 2009. (FM '09), p. 321–337. ISBN 978-3-642-05088-6. Disponível em: [http://dx.doi.org/10.1007/978-3-642-05089-3\\_21](http://dx.doi.org/10.1007/978-3-642-05089-3_21).
- PATIL, S. *Limitations and capabilities of Dijkstra's semaphore primitives for co-ordination among processes*. M.I.T., 1971. (Computation Structures Group memo // Massachusetts Institute of Technology, Project MAC). Disponível em: <http://books.google.com/books?id=GGqxGwAACAAJ>.
- ROSCOE, A. *The Theory and Practice of Concurrency*. [S.l.]: Prentice-Hall, 1998.
- SCATTERGOOD, B. *The semantics and implementation of machine-readable CSP*. University of Oxford, 1998. Disponível em: <http://books.google.com/books?id=MCrtIwAACAAJ>.
- SUN, J.; LIU, Y.; DONG, J. S. *Process Analysis Toolkit (PAT) User Manual*. 3.4. ed. [S.l.], 2007. Disponível em: <http://www.comp.nus.edu.sg/pat/OnlineHelp/index.htm>.
- SUN, J.; LIU, Y.; DONG, J. S. Model checking csp revisited: Introducing a process analysis toolkit. In: *In ISoLA 2008*. [S.l.]: Springer, 2008. p. 307–322.



- WOHLIN, C. *Experimentation in software engineering: an introduction*. Kluwer Academic, 2000. (Kluwer international series in software engineering). ISBN 9780792386827. Disponível em: <<http://books.google.com/books?id=nG2USHV0wAEC>>.

## *APÊNDICE A - Experimento E1 (Resultados)*

## *APÊNDICE B - Experimento E2 (Resultados)*

## *APÊNDICE C - Experimento E3 (Resultados)*

## *APÊNDICE D - Experimento E4 (Resultados)*