Uma Ferramenta de Suporte ao Ensino de Modelagem de Sistemas Distribuídos Críticos: Uma Experiência Prática

Edwin Juan Monteiro[1], Luis Rivero[2] e Raimundo Barreto[1]

[1]Instituto de Computação – Universidade Federal do Amazonas (UFAM)
Manaus – AM – Brasil
[2]DEINF - Programa de Pós-Graduação em Ciência da Computação
Universidade Federal do Maranhão
São Luis, Maranhão, Brasil

{ejlbm, rbarreto}@icomp.ufam.edu.br, luisrivero@nca.ufma.br

Abstract. *Software development teams should apply models that consider specific aspects of the problem domain in order to provide secure and reliable distributed systems. Although higher education institutions apply well-known models such as the Finite State Process (FSP), there are few support tools for teaching these models. In order to support the teaching of FSP, the fsp2java tool was developed, which allows the validation of the model generated by the students and the generation of Java code. This paper presents the development process of the tool and its empirical evaluation. The results demonstrate that the tool has the potential to act in the future teaching of these models.*

Resumo. *As equipes de desenvolvimento de software devem aplicar modelos que considerem aspectos específicos do domínio do problema, a fim de fornecer sistemas distribuídos confiáveis. Apesar de instituições de ensino superior aplicarem modelos conhecidos como o Processo de Estados Finitos (FSP), existem poucas ferramentas de apoio para o ensino destes modelos. Com o intuito de apoiar o ensino de FSP, foi desenvolvida a ferramenta fsp2java, que permite a validação do modelo gerado pelos alunos e geração de código Java. Este artigo apresenta o processo de desenvolvimento da ferramenta e sua avaliação experimental. Os resultados demonstram que a ferramenta tem potencial para futuramente atuar no ensino destes modelos.*

1. Introdução

Um sistema distribuído é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente [Tanenbaum e Steen 2006]. Neste tipo de sistema, um dos pilares para sua construção é o paradigma de programação concorrente. Dois ou mais processos cooperam para atingir um objetivo, em que cada processo é um conjunto de instruções executadas sequencialmente, como se fosse um programa sequencial [Lönnberg e Berglund 2007]. Devido à natureza concorrente deste tipo de sistemas, erros de programação podem ocasionar falhas na realização de tarefas. Por exemplo, o *Therac-25*, uma máquina de terapia com radiação, causou seis acidentes no período de dois anos devido a falhas de programação. O incidente causou overdoses massivas acarretando mortes e lesões sérias [Leveson e Turner 1993].

DOI: 10.5753/cbie.sbie.2018.218

Consequentemente, desenvolver sistemas distribuídos requer garantias de segurança verificadas e validadas.

Uma forma de evitar que o sistema distribuído não conterá falhas é através do uso de modelos de software. Um modelo de software possibilita simulações das variadas situações permitindo a verificação e localização de erros assim como a redução de custos com manutenção do software [Visser et al 2002]. Apesar da importância da modelagem em programação concorrente, o ensino deste paradigma é encarado com grande dificuldade por alunos de Ciência da Computação devido à falta de objetos de aprendizagem para apoiar o ensino.

Em seu livro, Magee e Kramer (2006) utilizam a linguagem de modelagem Processo de Estado Finito (FSP) como um modelo de computação distribuída para especificar o comportamento entre processos. Além disso, os autores implementaram a ferramenta Analisador de Sistema de Transição Rotulada (LTSA) para compilar as especificações FSP em máquinas de estado finito. A notação FSP é um modelo de aprendizagem com grandes recursos para a modelagem de sistemas distribuídos, dada a sua capacidade de representar diversos recursos como a concorrência, ações e dados compartilhados, recursão, incluindo a detecção de *deadlock* e de progresso, proporcionados por uma sintaxe enxuta e que evita detalhes desnecessários, como por exemplo, a alocação de memória e outros recursos. Ambas, a linguagem e a ferramenta, são usadas em cursos de computação para o ensino de programação concorrente. Entretanto, a ferramenta LTSA não indica com precisão os erros de sintaxe na modelagem. Por exemplo, um processo **P** seguido por um prefixo de ação "→" tem como mensagem de erro a ausência do parêntesis ")", contudo o usuário deveria ser notificado que o prefixo de ação não deve ocorrer após um processo. Exemplos como este dificultam que o aluno identifique e compreenda os erros cometidos nos modelos elaborados.

Considerando os problemas com a tecnologia acima, desenvolvemos o *fsp2java* [Monteiro et al 2018], que também é uma ferramenta para modelagem de processos concorrentes, com o diferencial que esta ferramenta permite identificar com precisão o local do problema de sintaxe assim como a execução do modelo. Além disso a ferramenta permite a codificação automática para a linguagem Java. Para a geração de código automático, o *fsp2java* segue um padrão de projeto, pois esses padrões capturam a *expertise* na construção de software orientado a objetos [Budinsky et al 1996] que alunos novatos ainda não têm, por exemplo, na programação concorrente. Para se obter respostas da eficácia da ferramenta no âmbito do ensino e a avaliação dos usuários acerca da sua utilização, executou-se um estudo no qual os alunos de Ciência da Computação responderam um questionário para avaliar a ferramenta de apoio à aprendizagem. Este artigo apresenta os resultados obtidos na avaliação da ferramenta e as oportunidades de melhoria identificadas.

Além desta introdução, o restante deste trabalho está organizado em quatro seções. A Seção 2 apresenta os trabalhos relacionados à métodos formais para o ensino da programação concorrente evitando que os alunos se preocupem com a sintaxe das linguagens de alto nível. A Seção 3 aborda o desenvolvimento do *fsp2java*. Na seção 4 é apresentada a avaliação do *fsp2java* cobrindo o planejamento e execução e encerrando

com a análise de resultados. Por fim, a Seção 5 apresenta as considerações finais e trabalhos futuros.

2. Trabalhos Relacionados

Aprender os conceitos de programação concorrente é essencial para estudantes de ciência da computação. Uma das formas mais comuns de programação concorrente é a que adota a programação com múltiplas *threads*. Contudo, a mudança de paradigma sequencial para *multithread* causa problemas significativos aos estudantes, pois as interfaces de programação geralmente são mais complexas do que o necessário, fazendo com que os alunos gastem tempo aprendendo os detalhes do sistema em vez dos fundamentos [Carr et al 2003].

Dentre as ferramentas de modelagem encontradas para o ensino de sistemas distribuídos, podemos citar: (a) SPIN, (b) SCML e (c) LTSA. O SPIN é um sistema de verificação eficiente para modelos de sistemas de software distribuídos. Ele foi usado para detectar erros de projeto em aplicativos que variam de descrições de alto nível de algoritmos distribuídos a códigos detalhados para controlar as centrais telefônicas [Holzmann 2004]. Por sua vez, o SCML é uma ferramenta que pode ser usada para simular um sistema de processos concorrentes que se comunicam através de variáveis compartilhadas. São suportados mecanismos para definir não determinismo, ações atômicas e sincronização de processos. Além disso, o SMCL inclui um protótipo para verificar propriedades básicas de segurança, como exclusão mútua e ausência de *deadlocks* usando a técnica de verificação de modelo [Ben-Ari 2007].

A ferramenta LTSA é utilizada em inúmeras universidades em todo o mundo junto com o livro de Magee e Kramer (2006) sobre concorrência [Lang 2010]. Esta ferramenta compila as especificações do FSP em uma máquina de estado e se assemelha com o autômato finito não determinístico de [Hopcroft e Ullman 1979]. O LTSA ainda permite visualizar e animar transições rotuladas através de interfaces gráficas. Ambos, FSP e LTSA são bastante utilizados.

Apesar do uso destas ferramentas para o ensino de concorrência [Lang 2010], vários alunos têm dificuldades no ensino, visto que os mesmos não são notificados quanto ao erro específico de sintaxe durante o aprendizado e/ou a importação do modelo para linguagens específicas de programação.Tendo em vista os problemas citados e a utilização do FSP em larga escala, foi proposto o *fsp2java*.

3. Desenvolvimento da ferramenta fsp2java

3.1 Definição do Escopo

Para este artigo, a construção da ferramenta teve o escopo do FSP reduzido aos seguintes construtores:

Prefixo de ação: A especificação **a → P** define que um processo executa uma ação **a**, e após o seu término o processo se comporta como descrito pelo processo **P**.

Escolha: Dadas duas ações **x** e **y** de um processo, então **(x→P | y→Q)** descrevem um processo que pode assumir a transição prefixada por **x** ou a ação prefixada por **y**.

Ação de guarda: A escolha **(when T x → P | y → Q)** implica que se **T** é verdadeiro então **x→P** é uma opção válida, **y→Q** não depende da condição para ser um caminho válido. Se **T** é falso, **x** não pode ser escolhido, **y** é a única ação possível.

Extensão de alfabeto: O alfabeto de um processo descreve as ações que ele pode executar. **P + C** estende o alfabeto de **P** com ações do conjunto **C**.

Composição paralela: Seja **P** e **Q** processos, **(P || Q)** representa a execução concorrente de **P** e **Q**.

Rotulação: A descrição **a:P** acrescenta o prefixo **a** em cada ação do alfabeto de **P**, desta forma a tem acesso as ações de **P**.

Renomeação de rótulo: A renomeação de rótulo é aplicada a um processo para alterar os nomes das ações. A forma geral de renomeação é: */ {rotuloNovo_1 / rotuloAntigo_1, ... rotuloNovo_n / rotuloAntigo_n}*.

A subseção seguinte explica a relação de mapeamento entre o escopo definido e a linguagem Java.

3.2 Mapeamento de FSP para Java

As associações essenciais para o entendimento são os processos e as suas respectivas ações. Os processos equivalem as classes, enquanto o alfabeto de ações corresponde aos métodos da respectiva classe. Por exemplo, vamos modelar o semáforo de trânsito. Em FSP a modelagem fica:

```
Semaforo = (vermelho→verde→laranja→Semafaro).
```

Neste modelo, após a ação laranja ser acionada, o processo é executado recursivamente para sempre. O código em Java para este modelo equivale a seguinte classe:

```
class Semaforo{
  public void vermelho(){}
  public void verde(){}
  public void laranja(){}
}
```

Para acessar os métodos da classe, uma instância com o mesmo nome do processo é definida. Para a classe semáforo, a instância fica *semaforo$* de forma que o acesso ao método vermelho, por exemplo, é *semaforo$.vermelho()*. O caractere **$** é utilizado para a distinção entre processos rotulados e não rotulados. Na ocorrência de um ou mais rótulos, por exemplo **{x, y}:Processo**, o **$** também permite a distinção entre rótulos através de uma instância para cada rótulo: *processo$x* ou *processo$y*. No exemplo do semáforo, a instância *semaforo$* indica que o processo não possui rótulo. A Tabela 1 contém alguns dos mapeamentos da notação FSP para a linguagem Java.

3.3 *fsp2java*

O *fsp2java* é uma ferramenta para a validação de modelos e geração de código automático que recebe como entrada uma notação textual FSP e retorna na saída o resultado da análise do modelo e o código em Java, caso a modelagem seja válida. Desta forma, uma análise sintática é aplicada no arquivo de entrada com a finalidade de garantir que o modelo fornecido atenda as especificações da notação. Caso a modelagem não respeite as características do FSP, uma mensagem de erro é exibida com uma breve

descrição da inconsistência contendo a linha e coluna que ocorre. Com a notação devidamente verificada, o *fsp2java* consulta o usuário para que um rastreamento (*trace*) finito seja definido em tempo de execução de modo interativo, pois, cada fluxo de ações de um ou mais processos pode conter caminhos não determinísticos.

Tabela 1. Relação entre a notação FSP e a linguagem Java.

FSP	Sintaxe	Java
Processo	P	class P { }
Ação	a → P	public void a (){ }
Escolha	(x → P | y → Q)	public void x (){ } public void y (){ }
Constante	const N	final int N
Ação de guarda	when(i < N)	if(i < N){ }
Processo STOP	a → STOP	return

O *trace* também delimita as ações recursivas de forma que o loop é encerrado à medida que satisfaz o fluxo de ações determinado pelo usuário, e assim pode-se encerrar a interpretação mecânica do FSP para que se inicie a geração do código equivalente em Java.

Para desenvolver a ferramenta definiu-se uma gramática EBNF (*Extended Backus-Naur*) que reconheça a notação FSP de forma que ela sirva como arquivo de entrada para o gerador de compilador Coco/R [Moessenboeck 1990]. Adotando a maneira de codificar feita por Magee e Kramer (2006), a semântica da codificação interpreta processos como classes e ações como métodos.

Para exemplificar o funcionamento do *fstp2java*, tomaremos como base o modelo de contagem infinita, *Count*, descrito abaixo:

```
Count (N = 3) = Count[0],
Count[i:0..N] = (
  when (i<N) inc → Count[i+1]
  | when(i>=0) dec → Count[i-1]
)
```

A Figura 1 representa a execução deste modelo no *fsp2java*. As ações {*inc, dec*} do processo *Count* são exibidas, uma por linha, para que o usuário escolha a ação que satisfaça a condição especificada. Desta forma, ocorre a mudança de estado do processo e a construção do *trace* finito. A figura ainda mostra o caminho disponível quando a variável i equivale a 0. É possível notar na figura o passo após o único caminho ser escolhido. A variável é incrementada de 1, e neste caso, a ação *dec* também torna-se elegível para escolha.

Na Figura 2, temos o código gerado com base no *trace* final, ações {*inc, inc*}, que permite a geração de dois arquivos Java. A figura exibe o arquivo *Count.java* que contém a classe *Count* e os todos os métodos, neste caso *inc* e *dec*. O código em (b) e (c) representa a classe principal do programa. Esta classe implementa a classe *Runnable* para execução dos métodos em uma *thread*. O arquivo ainda contém os atributos *objThread* e *count$* para acesso aos métodos da *thread* e de *Count*. O *trace* é executado

no método *run* de modo que cada método acessado por *count$* exibe na tela o seu respectivo nome respeitando as escolhas prévias do usuário.

```
Arquivo Editar Ver Pesquisar Terminal Ajuda
pc fsp2java # java -jar fsp2java.jar entra
da.txt
================================================
===========
TRACE ATUAL: []

Valor atual de i: 0
Valor atual de N: 3


CAMINHO DISPONIVEL em Count, ESCOLHA UMA OPCAO:

0 - inc->COUNT[i+1]

Digite -1 para sair
(a)
```

```
Arquivo Editar Ver Pesquisar Terminal Ajuda
===========
TRACE ATUAL: [count$.inc, count$.inc]

Valor atual de i: 2
Valor atual de N: 3


CAMINHO DISPONIVEL em Count, ESCOLHA UMA OPCAO:

0 - inc->COUNT[i+1]
1 - dec->COUNT[i-1]

Digite -1 para sair
(b)
```

Figura 1. (a) execução do FSP com apenas a ação *inc* habilitada; (b) i valendo 1, duas opções estão elegíveis, *inc* e *dec*.

```
Arquivo Editar Ver Pesquisar Terminal Ajuda
//Classe Count
class Count{
        public void inc(){
                System.out.println("inc");
        }
        public void dec(){
                System.out.println("dec");
        }
}
:
(a)
```

```
Arquivo Editar Ver Pesquisar Terminal Ajuda
//Classe Principal
import java.lang.Runnable;
import java.lang.Thread;
class MainClass implements Runnable{
        Thread objThread;
        Count count$;
        public void start(){
                count$ = new Count();
@
:
(b)
```

```
Arquivo Editar Ver Pesquisar Terminal Ajuda
                objThread = new Thread(this);
                objThread.start();
        }
        public void run(){
                try{
                        while(true){
                                count$.inc();
                                count$.inc();
                                return;
                        }
                }catch(Exception e){
                        e.printStackTrace();
                }
        }
        public static void main (String args [])
{
                MainClass objMainClass = new Mai
nClass();
                objMainClass.start();
        }
}
:
(c)
```

Figura 2. (a) classe *Count* para representar processo com as suas respectivas ações que agora são descritas como métodos; Em (b) e (c), a classe Principal executará os métodos de *Count*.

4. Avaliação do *fsp2java*

4.1. Planejamento e Execução

Alunos de 5° e 8° períodos do curso de Ciência da Computação da Universidade Federal do Amazonas (UFAM), utilizaram a ferramenta *fsp2java* para avaliar a viabilidade de uso em sala de aula. O Grupo foi composto por 5 alunos que detinham os conhecimentos sólidos (disciplinas cursadas e leitura de livros) sobre programação paralela e conhecimentos introdutórios de programação distribuída e métodos formais (participação em seminários). No exercício devia ser modelado o problema da máquina

de refrigerantes sem troco que aceitava apenas moedas de 5, 10 e 15 centavos. Os alunos submeteram o modelo ao *fsp2java* e avaliaram os avisos quando havia inconsistência na definição do FSP para correção. Por fim, executaram e avaliaram o código gerado.

Para estarem aptos a utilizar o *fsp2java*, os alunos participaram de uma aula sobre a notação FSP e utilização do *fsp2java*. Durante a experimentação, cada aluno utilizou computador individual e sem consulta. A participação na avaliação da ferramenta ocorreu em duas etapas: a) modelagem e teste da ferramenta; b) resposta do questionário.

Nesta esta avaliação foram elaborados dois questionários. O primeiro estava relacionado à percepção dos alunos sobre o modelo gerado pela ferramenta e o segundo estava relacionado à percepção dos alunos sobre o código gerado em java pela ferramenta. Nesse contexto, adaptou-se o modelo proposto por Martínez-Torres (2008) como base para avaliar a motivação de uso de ferramentas de modelagem de acordo com os seguintes aspectos: (a) intenção de uso, (b) facilidade de uso, (c) corretude, (d) confiança, (e) satisfação, e (f) utilidade. As Tabelas 2 e 3 apresentam os aspectos avaliados pelos participantes. Os alunos deviam informar o seu grau de concordância com o item apresentado para cada aspecto utilizando uma escala likert de 7 pontos: discordo totalmente, discordo amplamente, discordo parcialmente, neutro, concordo parcialmente, concordo amplamente e concordo totalmente.

Tabela 2. Itens levados em consideração para avaliar o modelo.

Item	Corretude	Confiabilidade	Facilidade	Qualidade	Satisfação	Utilidade
Descrição	Quando uso a **ferramenta de apoio,** a mesma **funciona corretamente para modelar** um sistema distribuído.	Eu **confio** na validação do modelo feita pela **ferramenta de apoio.**	A **ferramenta de apoio** é **fácil de usar para modelar** um sistema distribuído,	A **ferramenta de apoio** é **útil para modelagem** de sistemas distribuídos com qualidade.	Eu estou **satisfeito** com a validação do modelo feita pela **ferramenta de apoio.**	Eu usaria a **ferramenta de apoio** quando quisesse **modelar** sistemas distribuídos com qualidade.

Tabela 3. Itens levados em consideração para avaliar geração de código.

Item	Corretude	Confiabilidade	Facilidade	Qualidade	Satisfação	Utilidade
Descrição	Quando uso a **ferramenta de apoio,** a mesma **funciona corretamente para gerar código de** um sistema distribuído.	Eu **confio** no código gerado pela **ferramenta de apoio.**	A **ferramenta de apoio** é **fácil de usar para gerar código** de um sistema distribuído.	A **ferramenta de apoio** é **útil para gerar código** de um sistema distribuído com qualidade.	Eu estou **satisfeito** com código gerado pela **ferramenta de apoio.**	Eu usuaria a **ferramenta de apoio** quando quisesse **gerar código** de um sistema distribuído.

Além desta avaliação inicial, foram aplicadas seis perguntas abertas: a) "Qual é sua opinião quanto ao feedback da ferramenta para apoiar a sua aprendizagem/desempenho ao modelar sistemas distribuídos e/ou sistemas operacionais e/ou sistemas *multithreaded*?"; b) "O que dificultou o uso da ferramenta de apoio para

criar modelos baseados em Processo de Estados Finitos (FSP - *Finite State Process*)?"; c) "O que você mudaria no uso da ferramenta de apoio para melhorar a sua aprendizagem/desempenho ao modelar sistemas distribuídos e/ou sistemas operacionais e/ou sistemas *multithreaded*?"; d) "Qual é sua opinião quanto ao feedback da ferramenta para apoiar a sua aprendizagem/desempenho ao gerar código de sistemas distribuídos e/ou sistemas operacionais e/ou sistemas *multithreaded*?"; e) "O que dificultou o uso da ferramenta de apoio para gerar código fonte em Java de sistemas distribuídos e/ou sistemas operacionais e/ou sistemas *multithreaded*?"; f) "O que você mudaria no uso da ferramenta de apoio para melhorar a sua aprendizagem/desempenho ao gerar código de sistemas distribuídos e/ou sistemas operacionais e/ou sistemas *multithreaded*?". A seguir são apresentados os resultados obtidos na avaliação da ferramenta.

4.2. Análise dos Resultados

Os resultados obtidos do *fsp2java* para a modelagem de sistemas distribuídos indicam 4 alunos, confiam na ferramenta e que a mesma funciona corretamente, 3 acreditam que a ferramenta é fácil de usar. Vale ressaltar que ao somar o grupo de alunos que concorda totalmente e o grupo de alunos que concorda amplamente, 3 aprovam qualidade do modelo verificado, 4 julgam a ferramenta útil e estão satisfeitos com os resultados. A Figura 3 mostra a avaliação de cada item da Tabela 2 avaliado no questionário.

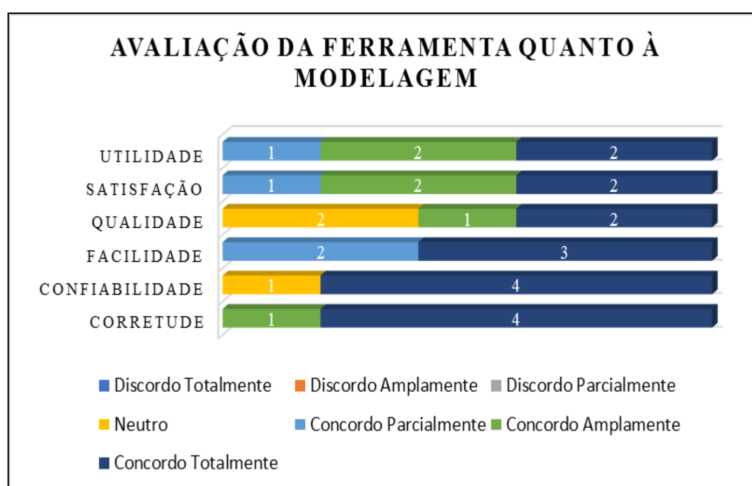

Figura 3. Gráfico de avaliação do *fsp2java* considerando os aspectos de modelagem.

Com respeito a geração de código, a Figura 4 mostra o percentual de cada um dos aspectos levados em consideração no questionário. Observa-se que 4 alunos (80%) julgam a ferramenta confiável de modo que o código gerado atende as expectativas a partir do modelo desenvolvido, tendo em vista a conformidade de 4 discentes. Ainda temos que 3 alunos acreditam que a ferramenta é de fácil utilização e que produz código Java de qualidade. Por fim, os 5 alunos estão satisfeitos com o código gerado. Os itens considerados nesta avaliação podem ser verificados na Tabela 3.

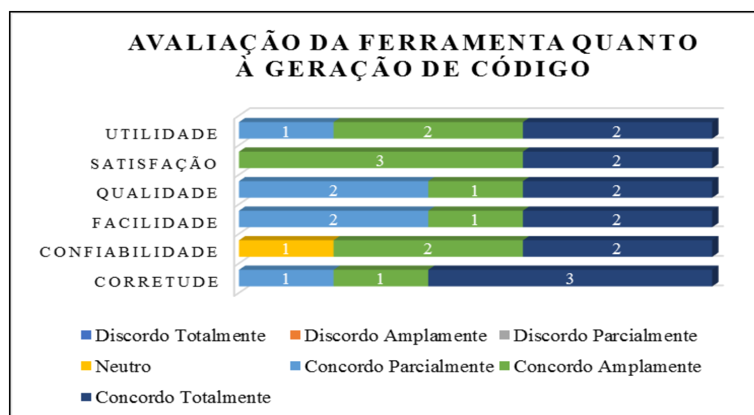

Figura 4. Gráfico de avaliação do *fsp2java* considerando os aspectos de geração de código automático.

Para entender melhor os aspectos que impactaram de maneira positiva ou negativa a compreensão dos alunos, foram analisadas as respostas abertas do questionário para identificar *feedbacks* que explicassem as notas atribuídas nos itens avaliados no questionário. A seguir são apresentados alguns comentários positivos e negativos considerando as respostas às questões abertas, onde Pi corresponde ao i-ésimo participante:

"A ferramenta é bastante intuitiva e de fácil uso. Os erros foram fáceis de achar e corrigir" – Positivo (P02).

"Depois de ter sido explicada, a ferramenta torna-se fácil de utilizar e gera o código corretamente quando se trata de modelagem" – Positivo (P04).

"A ausência de uma interface gráfica prejudica um pouco, mas a exibição dos traces compensa" – Negativo (P05).

"*A parte de geração de código deve ser mais transparente ao usuário*" – Negativo (P01).

5. Considerações Finais

Este artigo apresentou o *fsp2java*, uma ferramenta para modelagem de sistemas críticos. A ferramenta foi desenvolvida para atuar como objeto de aprendizagem em disciplinas que envolvam a programação concorrente, de forma que os alunos tenham um primeiro contato amigável com o paradigma e preocupem-se apenas em compreender os fundamentos. A dificuldade de dominar a sintaxe de uma linguagem de alto nível pode ser contornada pela funcionalidade complementar do *fsp2java,* a qual permite gerar código automático em Java. Após o uso do da ferramenta, foi aplicado um questionário que avaliou a percepção dos alunos em relação à confiabilidade, funcionalidade e corretude da ferramenta. Em termos de geração de código, a opinião dos alunos foi positiva para os aspectos de corretude, confiabilidade e utilidade. Apesar da satisfação entre os alunos com a ferramenta, é preciso tornar a mesma mais intuitiva para o uso.

Como trabalho futuro pretende-se incluir alguns dos pontos de melhorias citados pelos alunos nas respostas às questões abertas: (1) incluir interface gráfica para permitir uma interação mais harmoniosa com o usuário; (2) incluir um manual dentro da ferramenta que explique a sua utilização; (3) incluir de maneira transparente o

mapeamento de FSP para Java, pois ajudará o aluno a entender melhor o funcionamento do mesmo. Outro ponto de melhoria é aumentar o escopo atual do FSP presente na ferramenta para aceitar definições de prioridade em uma composição de processos além da sincronização dos mesmos.

Devido à avaliação do *fsp2java* ter sido realizada com poucos alunos, os resultados não podem ser generalizados. Como trabalho futuro, pretende-se: (a) aplicar uma nova avaliação aumentando o número de alunos e usando uma nova versão da ferramenta; (b) observar o impacto de uso da ferramenta dependendo do plano de aula aplicado; e (c) realizar um estudo comparativo para verificar o impacto da ferramenta. Desta forma, espera-se que o *fsp2java* seja eficaz como objeto de aprendizagem, a fim de auxiliar o aluno em seus estudos e contribuir com o ensino de sistemas distribuídos.

Referências

Ben-Ari, M. (2007). Teaching concurrency and nondeterminism with spin. ACM SIGCSE Bulletin, Volume 39.

Budinsky, F. J et al. (1996). Automatic code generation from design patterns. IBM systems Journal, p. 151-171.

Carr, S et al. (2003). ThreadMentor: a pedagogical tool for multithreaded programming. Journal on Educational Resources in Computing (JERIC), Volume 3.

Holzmann, G. J. (2004). The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Boston, MA.

Hopcroft, J. E e Ullman, J. D. (1979), "Introduction to Automata Theory, Languages, and Computation," Addison-Wesley, Reading, MA.

Lang, F. et al. (2010). Translating FSP into LOTOS and networks of automata. Formal Aspects of Computing, Vol: 22.

Leveson, N. G. e Turner, C. S. (1993). An investigation of the Therac-25 accidents. Computer, v. 26, n. 7, p. 18-41.

Lönnberg, J e Berglund, A. (2007). Students' understandings of concurrent programming. Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88. Australian Computer Society, Inc.

Magee, J. e Kramer, J. (2006). Concurrency: State Models & Java Programs. Wiley, 2nd edição.

Martínez-Torres, M. R. et al. (2008). A technological acceptance of e-learning tools used in practical and laboratory teaching, according to the European higher education area. Behaviour & Information Technology.Volume 27.

Moessenboeck, H. (1990). Coco/R: A Generator for Fast Compiler Front-Ends. ETH.

Monteiro, E. et al. (2018). fsp2java, Repositório GitHub, https://github.com/edwinlbm/fsp2java/tree/master/fsp, Julho.

Tanembaum, A. S e Steen M. V. (2006). Sistemas Distribuídos: Princípios e Práticas. Pearson, 2nd edição.

Visser, W et al. (2003). Model Checking Programs. Automated Software Engineering.