Evento: XXVI Jornada de Pesquisa

MODELAGEM E SIMULAÇÃO DO COMPORTAMENTO DO MOTOR DE EXECUÇÃO BASEADO EM TAREFAS DAS PLATAFORMAS DE INTEGRAÇÃO FUNDAMENTADO NAS REDES DE PETRI[1]

MODELING AND SIMULATION OF THE BEHAVIOR OF THE RUNTIME SYSTEM TASK-BASED OF INTEGRATION PLATFORMS BASED ON PETRI NETWORKS

Diogo Izequiel Rudell[2], Fabricia Carneiro Roos Frantz[3]

[1] Pesquisa realizada na Universidade Regional do Noroeste do Estado do Rio de Grande do Sul, Departamento de Ciências Exatas e Engenharias, Programa de Pós-Graduação em Modelagem Matemática e Computacional, Grupo de Pesquisa em Computação Aplicada (GCA) – Ijuí, RS.
[2] Discente da Pós-Graduação da Universidade Regional do Noroeste do Estado do Rio de Grande do Sul, Departamento de Ciências Exatas e Engenharias, Programa de Pós-Graduação em Modelagem Matemática e Computacional, Grupo de Pesquisa em Computação Aplicada (GCA) – Ijuí, RS, Brasil. diogo.rudell@sou.unijui.edu.br
[3] Docente da Universidade Regional do Noroeste do Estado do Rio de Grande do Sul, Departamento de Ciências Exatas e Engenharias – Ijuí, RS, Brasil. frfrantz@unijui.edu.br

RESUMO

É comumente encontrado nos ambientes empresariais o uso de diversos *softwares*, assim como ecossistemas de *softwares* compostos por diversas tecnologias, modelos de dados e sistemas operacionais. Essa heterogeneidade impulsionou o campo de estudo da Integração de Aplicações Empresariais que busca fornecer metodologias e ferramentas para projetar e implementar uma solução para integrar um ecossistema de *software* heterogêneo. Existem diversas plataformas que foram desenvolvidas a fim de dar suporte para a integração de aplicações, fazendo com que as aplicações trabalhem de forma integrada. A execução de uma solução de integração é realizada pelo motor de execução das plataformas de integração, o que torna seu desempenho uma questão extremamente importante. Nesse sentido, este artigo modela o funcionamento do motor de execução baseado em tarefas das plataformas de integração utilizando as Redes de Petri e realiza experimentos, com a intenção de avaliar se a variação da quantidade de recursos computacionais influencia proporcionalmente no desempenho do motor de execução da plataforma de integração para executar uma solução sob uma determinada taxa de entrada de mensagens. Os resultados obtidos foram satisfatórios com relação ao modelo de simulação implementado e a variação dos recursos computacionais, que influenciam na performance do motor de execução.

Palavras-chave: Plataformas de Integração. Modelagem Matemática. Motor de execução baseado em tarefas. Redes de Petri.

ABSTRACT

It is commonly found in business environments the use of different software, as well as software ecosystems composed of different technologies, data models and operating systems. This heterogeneity boosted the field of study of Enterprise Application Integration, which seeks to

provide methodologies and tools to design and implement a solution to integrate a heterogeneous software ecosystem. There are several platforms that have been developed in order to support application integration, making applications work seamlessly. The execution of an integration solution is performed by the runtime system of the integration platforms, which makes its performance an extremely important issue. In this sense, this article models the operation of the tak-based runtime system of integration platforms using Petri Nets and performs experiments, with the intention of evaluating whether the variation in the amount of computational resources proportionally influences the performance of the runtime system of the platform. integration to run a solution under a given message input rate. The results obtained were satisfactory in relation to the implemented simulation model and the variation of computational resources, which influence the performance of the runtime system.

Keywords: Integration Platforms. Mathematical Modeling. Runtime System Task-Based. Petri Nets.

INTRODUÇÃO

A utilização de aplicações (*softwares*) pelas empresas ocorre devido a necessidade de competitividade, informatização e da necessidade de agilidade em seus processos de negócios, sendo que um *software* tem como finalidade organizar um conjunto de dados e informações de forma integrada. Com a intenção de atender as necessidades da empresa como coletar, organizar, distribuir e disponibilizar informações para os seus processos de negócios, ocorreu a aquisição e o desenvolvimento de aplicações em quantidade demasiada formando um ecossistema de *software* heterogêneo (MANIKAS, 2016).

A Integração de Aplicações Empresariais (EAI) busca oferecer metodologias, técnicas e ferramentas para a implementação de soluções de integração para ecossistemas de *software* (HOHPE, WOOLF, 2004). As plataformas de integração foram desenvolvidas para fornecer suporte para projetar, implementar, executar e monitorar as soluções de integração, fazendo com que as aplicações trabalhem de forma sincronizada. Pode-se citar dentre as várias plataformas de integração existentes a *Apache Camel* (IBSEN, ANSTEY, 2018), *Mule* (DOSSOT et al., 2014), *Spring Integration* (FISHER et al., 2012), *Petals* (SURHONE et al., 2010) e Guaraná (FRANTZ et al., 2011; FRANTZ et al. 2016; FRANTZ et al.2020).

Criada por pesquisadores que atuam no Grupo de Pesquisa em Computação Aplicada (GCA) da Universidade Regional do Noroeste do estado do Rio Grande do Sul - UNIJUÍ, a tecnologia Guaraná possibilita projetar, implementar e executar soluções de integração (FRANTZ et al., 2011; FRANTZ et al. 2016; FRANTZ et al.2020). A plataforma Guaraná segue o estilo de integração baseado em mensagens e dá suporte ao uso dos padrões de

integração. Além disso é construída usando a tecnologia Java e fornece uma linguagem de domínio específico, Guaraná DSL, um toolkit de desenvolvimento, um ambiente para realização de testes, uma ferramenta de monitoramento e um motor de execução (*Runtime System*) baseado em tarefas (*Task-Based*) (FRANTZ et al. 2016; BLYTHE et al., 2005; ALKHANAK et al., 2016).

O motor baseado em tarefas das plataformas de integração tem como objetivo principal executar tarefas, ou seja, executa os padrões que foram programados, modificando, transformando e roteando se necessário. O processo de execução de uma solução de integração pelo motor de execução baseado em tarefas se inicia quando uma mensagem chega na porta de entrada, e percorre o canal até a primeira tarefa do fluxo. Então cria-se uma anotação (*work unit*) que informa aos recursos computacionais (*threads*) que está pronta para ser executada, essa anotação vai para uma fila de trabalho (*work queue*),, que segue a disciplina de quem primeiro chega, primeiro sai (*first-in first-out*), também conhecida como FIFO, onde aguarda até ser executada pelos recursos computacionais. Ao terminar a execução na primeira tarefa a mensagem segue o fluxo até a próxima tarefa, onde se repete a mesma situação, como se pode observar na Figura 1. Desta forma, o processo faz com que exista a relação de dependência durante a execução das tarefas, onde a tarefa posterior só pode ser executada quando a primeira já tenha sido executada. Assim pode-se afirmar que o sistema de execução do motor de execução baseado em tarefas é totalmente assíncrono e sua arquitetura contém elementos que executam e registram o funcionamento do modelo (FRANTZ et al., 2011; FRANTZ et al. 2016; FRANTZ et al.2020).

A sequência de operações necessárias para iniciar os *threads* ocorre de forma síncrona ao chegar a *work unit* para processar. A lógica de funcionamento de uma *thread* é definida dentro de sua operação do trabalho quando uma *work unit* chega na *work queue* ela é analisada. A *thread* primeiro verifica seu tempo de execução agendado, se expirou, pode ser executada imediatamente, caso contrário, a *work unit* será adiada até o prazo final. Essa estratégia permite que os *threads* continuem trabalhando enquanto houver uma tarefa pronta para ser executada, ou seja, os *threads* vão consumindo tarefas da *work queue* (ALKHANAK et al., 2016).

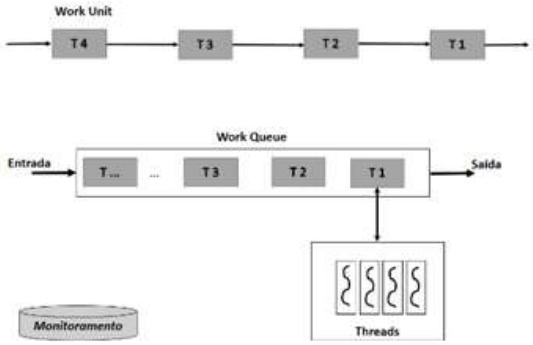

Figura 1. Representação do motor de execução baseado em tarefas das plataformas de Integração. Fonte: FRANTZ et al. 2020.

Para simular uma solução de integração faz-se necessário a utilização de ferramentas de simulação, que por consequência exigem a criação de um modelo de simulação a partir do modelo conceitual. Para construir modelos de simulação existem diversas linguagens e formalismos matemáticos, como por exemplo, as Redes de Petri, que permitem representar sistemas concorrentes, paralelos, assíncronos e não determinísticos (MURATA, 1989).

Como ferramenta gráfica, as Redes de Petri podem ser usadas como um auxílio na comunicação visual. Sendo semelhante a gráficos de fluxo, diagramas de blocos e redes (PETRI, 1962), permitindo uma visualização estrutural e comportamental dos processos do sistema (LAW, KELTON, 2000). Composta por dois elementos, as Redes de Petri são grafos bipartidos, onde os lugares representam uma condição, atividade ou recurso, sendo representados por círculos, e as transições representam um evento (ação) do sistema, cuja representação gráfica é um traço, barra ou retângulo. Enquanto os arcos que interligam transições aos lugares representam a relação entre as ações e as condições que se tornam verdadeiras com a execução das ações (PETERSON, 1977). As Redes de Petri oportunizam uma visualização de alta abstração e acompanhamento dinâmico das diferentes atividades do sistema que possuem características predominantemente discretas (AALST, 2015).

METODOLOGIA

A artigo segue algumas etapas a fim de promover mais visibilidade das informações e do processo de desenvolvimento do modelo de simulação, verificação e experimentação.

1. Descrição do problema: busca-se detalhar as variáveis envolvidas, seus relacionamentos e os parâmetros de entrada foram analisados e estudados.

2. Formulação do modelo: formula-se o modelo, visando compreender o problema de integração, para então desenvolver o modelo conceitual.

3. Implementação do modelo: o modelo é implementado, para isso é necessário desenvolver o modelo computacional com uma técnica de representação, Redes de Petri.

4. Verificação e experimentação: é realizada a verificação e a experimentação do modelo, na qual é preciso analisar e interpretar os resultados obtidos através da simulação do modelo computacional, além de transcrever as conclusões.

MODELO DE SIMULAÇÃO

O motor de execução das plataformas de integração é parte fundamental num processo de integração, pois é por meio dele que a execução acontece. Pode-se dividir o modelo de execução do motor em três partes: entrada, processamento e saída. A entrada corresponde a chegada das mensagens na porta de entrada do processo de integração. O processamento inicia quando a mensagem chega na porta de entrada e percorre o canal até encontrar a primeira tarefa do fluxo. Ao chegar na tarefa, o motor de execução cria uma anotação de que a mensagem está pronta para ser executada. A anotação é armazenada na fila de *work units*, a partir da qual as *threads* à processam, e a encaminham novamente para o fluxo. Então, esta mensagem percorre o fluxo até a próxima tarefa, dando sequência ao processo. Ao ser anotada a mensagem da última tarefa do processo, a mensagem é encaminhada para a saída.

Para construir o modelo de simulação foi utilizado uma técnica de representação que permite representar sistemas concorrentes, paralelos, assíncronos e não determinísticos, que são as Redes de Petri. As redes de Petri oportunizam uma visualização de alta abstração e acompanhamento dinâmico das diferentes atividades do sistema que possuem características predominantemente discretas (AALST, 2015). Sendo que uma Rede de Petri pode representar um sistema de transição de estados, onde os lugares são os estados e as transições são os eventos. Ao ocorrer um disparo esse sistema troca de estados, o que possibilita realizar uma análise sobre a execução do sistema. Uma grande variedade de ferramentas de simulação com linguagem de simulação, como SLAM, GPSS, GASP, POWERSIM, ARENA, EXTEND, PIPE2 e CPN *Tools*, foram introduzidas no mercado favorecendo a aplicação da simulação de

uma forma geral. Essas ferramentas combinam um ambiente de design gráfico e uma linguagem de programação como FORTRAN, C e PASCAL, além de oferecer recursos de análise gráfica e animação (LAW, KELTON, 2000; AALST, 2015). O CPN *Tools* permite a edição, simulação e análise de modelos de Redes de Petri Coloridas, o que oportuniza uma visualização de alta abstração e acompanhamento dinâmico das diferentes atividades do sistema (AALST, STAHL, 2011).

Na busca de embasamento técnico e científico encontrou-se um problema apresentado por Aalst, Stahl (2011), para a modelagem do funcionamento das *threads*, o problema da máquina de cartão perfurado. Problema este que pode apresentar mais *tokens* no local "*in operation"* e *"free"* e a máquina pode apresentar os estados de *"free"* e "*in operation"*, ou seja livre e ocupada. Desta forma, se fundamentou a disposição dos lugares e transições que representam os recursos computacionais no modelo de simulação, quanto à disposição dos arcos, lugares, transições e funções, como pode-se observar na Figura 2.

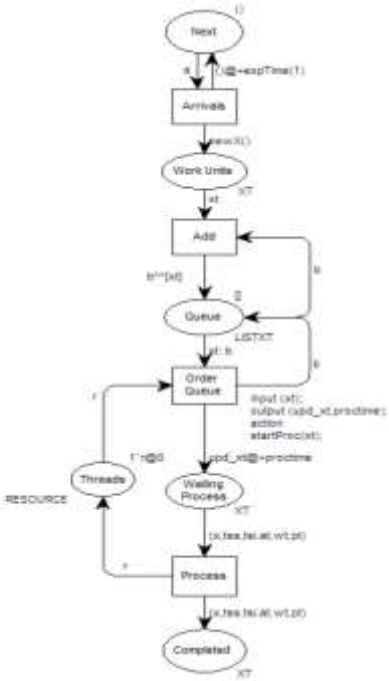

Figura 2. Modelo de simulação do motor de execução baseado em tarefas em Redes de Petri. Fonte: do autor.

A Figura 2 representa o funcionamento do motor baseado em tarefas em Redes de Petri. O processo se inicia com uma mensagem no lugar *Messages* que representa uma

mensagem na entrada do processo de integração e uma mensagem do lugar *Threads* que representa uma *thread*. A chegada de uma mensagem na primeira tarefa é representada pela transição *Arrival on task*, pois ocorre um evento, que é a criação da *work unit*, a qual é armazenada no local *Work Unit*.

Quando a transição *Arrival on task* dispara, a mensagem que se encontra no lugar *Messages* se desloca para o lugar *Work Unit* e uma nova mensagem é criada no lugar *Messages*, ou seja, a transição *Arrival on task* reproduz a chegada das mensagens no processo.

Dessa forma, a cada *work unit* criada a partir da mensagem de entrada, uma nova mensagem é introduzida no lugar de entrada, não obedece necessariamente uma ordem, ou seja a entrada pode ser variável ou aleatória. No CPN *Tools* a entrada aleatória pode ser representada com as funções de distribuição aleatórias que modelam com precisão atrasos de tempo. Esses atrasos de tempo podem ser distribuídos de forma exponencial utilizando a função descrita abaixo, que pode ser utilizada para criar tempos entre chegadas que são distribuídas de forma aproximadamente exponencial (AALST, STAHL, 2011).

```
fun expTime (mean: int) =
let
val realMean = Real.fromInt mean
val rv = exponential ((1.0/realMean)
in
floor (rv + 0.5)
end;
```

Cada lugar da rede possui associado um conjunto de cores como atributo, assim, ele só pode conter fichas daquele conjunto de cores específico. Assim como variáveis e funções são usadas como inscrições de transições e arcos. As declarações são parte integrante da rede e estão situadas na área de índice. Declarações do usuário podem ser adicionadas após as declarações padrão do CPN *Tools*, alguns conjuntos de cores e variáveis utilizadas estão descritas abaixo:

```
colset UNIT = unit timed;
colset X = with A | B;
colset TS = string;
colset TI = int;
```

*colset XT = product X * TS * TI * TI * TI * TI timed;*

colset LISTXT = list XT;

var a: UNIT;

var b: LISTXT;

O conjunto de cores *UNIT* é usado para representar um valor de carimbo de data/hora no local *Messages*, o qual é um conjunto padrão do CPN *Tools*. O conjunto de cores *TI* é usado para representar valores de tempo, incluindo carimbos de data/hora, como números inteiros. O conjunto de cores *X* descreve que há dois tipos de objetos: A e B. No modelo de simulação, uma mensagem é representada por um valor do conjunto de cores *XT*. Onde *XT* é um conjunto de cores de produto com 6 componentes(6-tupla), que representam:

- O tipo de objeto, ou seja, A ou B (*x*);
- O carimbo de data/hora do *token* representado como uma *string* (*tss*);
- O carimbo de data/hora do *token* representado como um número inteiro (*tsi*);
- A hora de chegada do objeto, ou seja, a hora em que o objeto chegou ao sistema (*at*);
- A quantidade total de tempo que o objeto deve esperar antes de ser processado pelos recursos (*wt*);
- A quantidade total de tempo que o objeto foi processado por recursos (*pt*).

Estes parâmetros são declarados no CPN *Tools* da seguinte forma:

var tsi, at, wt, pt, proctime : TI;

var tss : TS;

Quando ocorre a transição de chegadas *Arrival on Task*, a função *newX* é usada para criar a *work unit* que representa a mensagem e sua tarefa. Quando chamada, a função *newX* retorna um valor do conjunto de cores *XT*, ou seja, ela retornará uma 6-tupla, conforme descrita abaixo:

fun newX ( ) =

X.ran ( ),

ModelTime.toString (time ( )),

intTime ( ),

intTime ( ),

0,

0)

O primeiro componente é o tipo do objeto que é escolhido aleatoriamente usando a função *X.ran( )*. O segundo e o terceiro componentes são representações de *string* e inteiros do registro de data e hora da mensagem que é adicionada para colocar no lugar *Messages*. Uma vez que não há uma inscrição de tempo na transição nem um atraso de arco no arco do lugar para a transição, nem vice-versa, o carimbo de data/hora da mensagem adicionada ao lugar *Messages* será igual à hora do modelo em que as chegadas ocorrem. Quando a expressão *ModelTime.toString(time( ))* for avaliada, ela retornará a representação de *string* do tempo do modelo atual. A função *ModelTime.toString* é uma das funções do simulador.

O quarto componente representa o tempo de chegada da mensagem, como o horário de chegada é igual ao horário do modelo no qual as chegadas ocorrem, a função *intTime* também é usada para obter o horário de chegada da mensagem. O quinto e o sexto componentes representam os tempos totais de espera e processamento, respectivamente, para a mensagem. Esses valores são inicializados em 0.

fun intTime( ) = IntInf.toInt (time( ));

Após a criação da *work unit*, a mensagem passa a ser a variável *xt*, que é encaminhada para uma fila de *work units*, representada pelo lugar *Queue*. A marcação inicial do lugar *Queue* é uma marcação vazia [ ]. Com o propósito de obter uma lista FIFO, o conjunto de cores utilizado é o *LISTXT*, que corresponde a uma lista de inteiros e a variável *b* corresponde à lista. Funções padrão permitem acessar o primeiro e o último elemento de uma lista. Para acessar elementos do interior da lista, funções recursivas têm que ser usadas, como a que concatena o elemento *xt* na lista *b* (*b ^^ [xt]*) e a que coloca o elemento *xt* na cabeça da lista *b* (*xt :: b*) .

var xt, upd_xt : XT;

Quando a *work unit* está pronta para ser processada e é a primeira da fila, a transição *Order Queue* dispara. A transição *Order Queue* retira a *work unit* da fila e anexa uma *thread* que está disponível no local *Threads* para realizar o processamento no local *Waiting Process*. Quando ocorre o disparo da transição *Order Queue* vários dos atributos de tempo associados a *work unit* precisam ser atualizados, pois é acrescentado o tempo de processamento. A função *startProc* é chamada no segmento de código para a transição *Order Queue*, sendo usada para atualizar os atributos de tempo apropriados para cada *work unit*. A função *startProc* é definida da seguinte forma:

fun startProc ((x, tss, tsi, at, wt, pt): XT) =

```
let
val proc_time = expTime (avg_proc_time)
val time_stamp =
ModelTime.add (time ( ),ModelTime.fromInt (proc_time))
val new_tss = ModelTime.toString (time_stamp)
val new_tsi = IntInf.toInt (time_stamp)
val new_wt = wt + (intTime ( ) - tsi)
val new_pt = pt + proc_time
in
((x, new_tss, new_tsi, at, new_wt, new_pt),proc_time)
end
```

A função recebe um valor *xt* como argumento, que é uma *work unit* em início de processamento. Quando a transição *Order Queue* dispara os parâmetros de tempo da *work unit* são atualizados. Pois os *threads* levam tempo para processar uma *work unit*, então um atraso de tempo será adicionado a *work unit* que é adicionada ao local *Waiting Process*. Na função *startProc*, o valor *proc_time* é o tempo de processamento gerado para a *work unit*. O tempo de processamento é obtido chamando a função *expTime*, que é utilizada para criar tempos distribuídos de forma aproximadamente exponencial. Sendo que esses tempos possuem média de 1.

```
val avg_proc_time = 1;
```

O valor *proc_time* é usado para calcular a data e hora da *work unit*. Quando a transição *Order Queue* ocorre, o carimbo de hora para a *work unit* que é adicionado ao local *Waiting Process* será igual a (a hora do modelo em que a transição ocorre) + (o valor de *proc_time*, conforme a regra de ocorrência das CPNs cronometradas (ALLST, STAHL,2011)). Cabe salientar que o registro de data e hora da *work unit* a ser adicionada ao local *Waiting Process* será diferente do tempo do modelo atual, todavia a sequência de caracteres e as representações de inteiros do registro de data e hora da *work unit* devem ser atualizadas de acordo.

Além disso, enquanto não há recursos computacionais para processar a *work unit*, esta aguarda até haver *threads* disponíveis. Isso significa que o atributo de tempo que representa o tempo total de espera pela *work unit* deve ser atualizado. O novo tempo de espera total é igual

ao tempo de espera total anterior mais a quantidade de tempo que a *work unit* estava no local *Queue*.

Os parâmetros *tss* e *tsi* para a função *startProc* poderá ser usada para calcular a quantidade de tempo que a *work unit* permaneceu na fila. Quando uma *work unit* é removida da *Queue*, o tempo de espera mais recente para o objeto pode ser obtido subtraindo a representação inteira do carimbo de data/hora do tempo do modelo em que a transição ocorre, que é exatamente o que a seguinte expressão *startProc* faz: (intTime( ) - tsi). Assim, o valor local *new_wt* é o tempo de espera total atualizado corretamente para o objeto.

O tempo total de processamento também deve ser atualizado quando uma *work unit* é movida da fila para o lugar *Waiting Process*. Conforme descrito acima, o valor *proc_time* representa a quantidade de tempo em que a *work unit* será processada, e esse valor deve ser adicionado ao tempo total de processamento. É exatamente assim que o valor *new_pt* em função *startProc* é obtido. Observa-se que a função *startProc* não modifica o tipo de objeto ou o atributo de tempo de chegada do objeto, ou seja, os valores vinculados aos parâmetros *x* e *at* são retornados inalterados.

Os recursos computacionais são implementados no CPN *Tools* com a cor *RESOURCE* em um lugar nomeado como *Threads*, com a marcação *1`r@0*, de forma paralela com o lugar *Waiting Process*, conforme Aalst, Stahl (2011) salientou no problema da máquina de cartão perfurado.

colset RESOURCE = with r timed;

O CPN *Tools* disponibiliza recursos de monitoramento que podem ser usados para observar, inspecionar, controlar ou modificar um modelo de simulação. Onde cada medida de desempenho é calculada por um monitor coletor de dados e cada monitor coletor de dados pode calcular mais de uma medida de desempenho de interesse salvando os dados em arquivos. No modelo a quantidade de mensagens de entrada foi controlada através de dois monitores, *Arrived* e *NumArrived*.

O monitor *Arrived* é um monitor de ocorrências de transição de contagem, que calcula o número de vezes que a transição *Arrivals* ocorre durante uma simulação. O monitor *NumArrived* é um monitor de ponto de interrupção, que foi usado para interromper uma simulação quando atingir a quantidade pré-definida de mensagens que passaram na transição *Arrivals an task* em que está vinculado. A função de predicado utilizada no monitor

NumArrived acessa a estatística de contagem do monitor *Arrived* sempre que a transição *Arrivals an task* ocorre. Na função de predicado apresentada abaixo a contagem é igual a 100, ou seja, quando o monitor *Arrived* contar 100 mensagens, a função de predicado retornará *true* e a simulação será interrompida.

```
fun pred (bindelem) =
let
fun predBindElem (Model'Arrivals (1, {a} )) =
Arrived.count( ) ≥ 100
predBindElem_ = false
in
predBindElem bindelem
end
```

VERIFICAÇÃO E EXPERIMENTAÇÃO

Antes de simular e analisar o modelo de simulação, o CPN *Tools* permite realizar uma análise de sintaxe, simulação passo-a-passo e de propriedades através da análise do espaço de estados. O modelo não apresenta erros de sintaxe, como bolhas de conversação, auras ou bolhas de status com erros. A simulação passo-a-passo, cuja finalidade é identificar possíveis erros de desenvolvimento do modelo proposto, foi realizada com o auxílio de especialistas da área, comprovando a validade frente a finalidade e a representatividade do modelo proposto.

O espaço de estado foi gerado com o modelo ilustrado na Figura 2, com três mensagens de entrada e um *thread*, o qual resultou em 30 nós, 48 arcos, foi construído em menos de um segundo e contém todas as marcações alcançáveis. As estatísticas para o gráfico SCC também são especificadas, tendo 30 nós e 48 arcos, e foi calculado em menos de um segundo. A marcação inicial implementada faz com que o modelo seja não reiniciável, mas todas as transições são vivas segundo o relatório.

A *Home Properties* define que existe uma única marcação *Home Markings*, marcação inicial, no nó 30. Uma marcação *Home Markings* é uma marcação inicial que pode ser alcançada a partir de qualquer marcação alcançável. Em outras palavras, não podemos fazer coisas que tornem impossível chegar a *Home Markings* posteriormente (JENSEN,2007).

O modelo também possuí uma única marcação de terminação (*Dead Markings*), no nó 30. Esta é a marcação em que o modelo concluiu com sucesso a transmissão de todas as mensagens de entrada. O fato de a marcação *Dead Markings* e a marcação *Home Markings* serem iguais faz com que ao final da simulação o modelo retorna ao seu estado inicial.

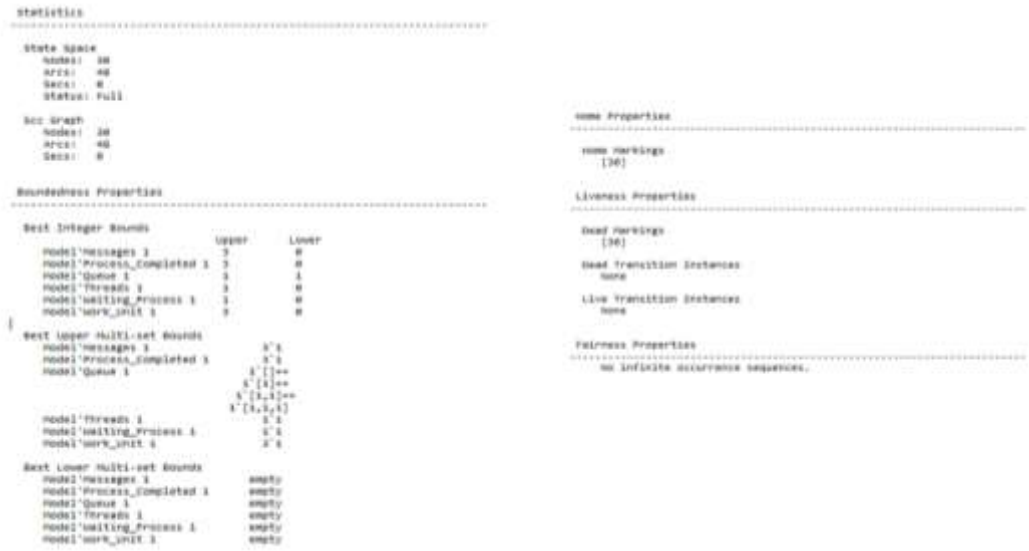

Figura 3. Relatório de espaço de estado do modelo de simulação. Fonte: do autor.

Os experimentos foram realizados em uma máquina virtual equipada com 2 processadores Intel(R) Xeon(R) CPU E5-4610 v4, 1.80GHz, 8 GB de RAM, e sistema operacional Windows 10 Education, versão 20H2. O CPN *Tools*, versão 4.0.1, foi instalado para executar a simulação. No estudo experimental foram executados 150 cenários, sendo 10 faixas de entrada (1300, 2.600, 3.900, 5.200, 6.500, 7.800, 9.100, 10.400, 11.700 e 13.000) e 15 diferentes faixas de recursos computacionais (1 a 15 *threads*), cada cenário foi repetido 30 vezes. A função *CPN'Replications.nreplications 30* foi usada para executar automaticamente 30 simulações (GEHLOT, NIGRO, 2010).

Ao final de uma simulação no CPN *Tools* é realizada a execução da função de replicação, que geram relatórios automaticamente. Conhecidos como relatórios de desempenho de replicação que contêm estatísticas que são calculadas para os valores de dados encontrados nos arquivos de cada replicação. O relatório fornece uma visão geral das estatísticas que são calculadas durante uma simulação para os monitores coletores de dados de uma rede.

Cada valor apresentado na tabela da Figura 4 representa a média calculada para as 30 repetições em cada cenário de entrada e número de *threads*. Os valores destacados em negrito representam

os cenários que possuem as médias de tempos totais de espera das mensagens com melhor performance. Considerando que a performance do motor de execução baseado em tarefas está relacionado a economia do uso de recursos e menor tempo de espera das mensagens processadas.

Threads	50 pedidos	100 pedidos	150 pedidos	200 pedidos	250 pedidos	300 pedidos	350 pedidos	400 pedidos	450 pedidos	500 pedidos
	1.300 Work Units	2.600 Work Units	3.900 Work Units	5.200 Work Units	6.500 Work Units	7.800 Work Units	9.100 Work Units	10.400 Work Units	11.700 Work Units	13.000 Work Units
1	26,356174	36,660345	59,370454	63,760391	76,533327	65,149398	63,240379	70,445340	82,144863	85,432832
2	0,457535	0,486576	0,454832	0,480436	0,466398	0,474327	0,465256	0,487862	0,479355	0,479353
3	0,090331	0,087354	0,087592	0,085946	0,089949	0,087634	0,089295	0,088996	0,090539	0,090769
4	0,019707	0,019901	0,022928	0,020637	0,022341	0,021211	0,022196	0,022038	0,021822	0,021331
5	0,005162	0,005748	0,005131	0,005463	0,005351	0,005527	0,005880	0,005473	0,005950	0,005686
6	0,001438	0,001360	0,001214	0,001969	0,001421	0,001505	0,001348	0,001327	0,001345	0,001536
7	0,000360	0,000308	0,000453	0,000789	0,000318	0,000406	0,000520	0,000478	0,000459	0,000403
8	0,000000	0,000115	0,000077	0,000096	0,000015	0,000113	0,000114	0,000128	0,000049	0,000115
9	0,000077	0,000013	0,000000	0,000071	0,000005	0,000017	0,000044	0,000006	0,000014	0,000023
10	**0,000000**	**0,000000**	**0,000000**	**0,000000**	**0,000000**	**0,000000**	0,000004	0,000000	0,000003	**0,000000**
11	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000004	0,000010	0,000000	0,000000
12	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	**0,000000**	**0,000000**	0,000003	0,000000
13	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000010	**0,000000**	0,000000
14	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
15	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000

Figura 4. Média das 30 repetições do tempo total de espera das *work units* durante a simulação. Fonte: do autor.

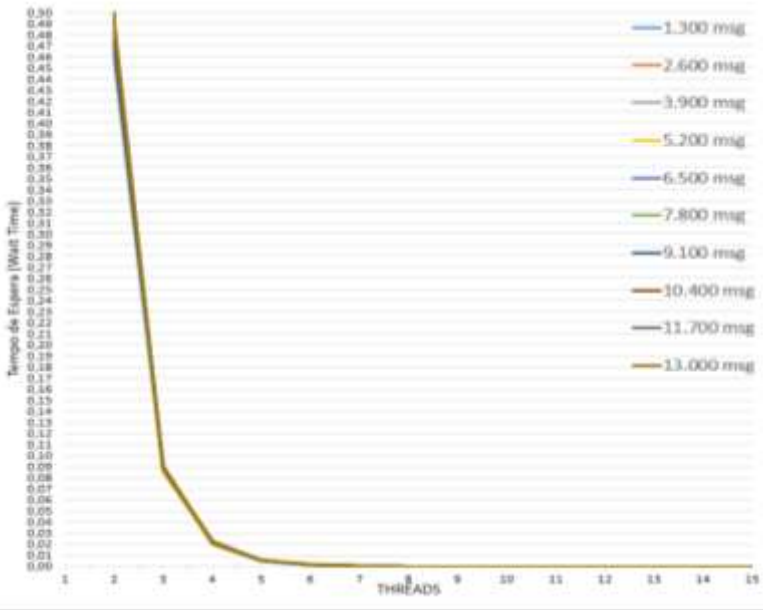

Figura 5. Gráfico das médias de tempo total de espera das *work units* no modelo durante a simulação. Fonte: do autor.

Com a entrada de 1.300 mensagens, a menor média de tempo total de espera ocorre com a configuração de 10 *threads*. Tornando as configurações com mais recursos

computacionais dispendiosas em relação a performance. O mesmo acontece com as entradas de 2.600, 3.900, 5.200, 6.500, 7.800 e 13.000 mensagens, onde a configuração com 10 *threads* foi a que obteve maior performance no uso dos recursos computacionais. Para as entradas de 9.100 e 10.400 mensagens, a eficiência na média de tempo total de espera foi normalmente atingida usando 12 *threads*.

No gráfico da Figura 5 o eixo x representa o número de *threads* e o eixo y a média de tempo total de espera das mensagens processadas. Percebe-se, no gráfico, que todos os resultados obtidos apresentam comportamentos semelhantes. Observa-se que no modelo existe uma maior espera das mensagens para valores de entradas mais altas, assim como a quantidade de *threads* necessárias para garantir uma boa performance deverá ser maior.

CONSIDERAÇÕES FINAIS

O modelo desenvolvido com Redes de Petri pode ser utilizado para auxiliar os engenheiros de *software* na configuração do motor de execução de uma plataforma de integração, e com isso, melhorar o desempenho de uma solução de integração, substituindo a necessidade do conhecimento empírico. O modelo permite conhecer previamente os limites de uma solução de integração, por exemplo, se o engenheiro de *software* tiver apenas um certo número de *threads* disponíveis em seu computador, é possível por meio de experimentos descobrir a taxa de entrada máxima suportada para obter um bom desempenho.

O relatório de espaço de estados gerado no CPN *Tools* comprovou com as propriedades estatísticas, limitabilidade, vivacidade, imparciabilidade e de conservabilidade que o modelo não possui erros e contém as características necessárias para representar de forma satisfatória o funcionamento do motor de execução baseado em tarefas da plataforma Guaraná. Ao realizar a análise dos dados dos experimentos, é possível observar que o motor de execução possui um desempenho com grande regularidade para pequenas entradas. O modelo de simulação apresentou durante as simulações bons resultados, possibilitando a seguinte análise, a utilização de dois ou mais *threads* fazem com que a média de tempo total de espera seja menor que 0,5 e em todos os cenários o uso demasiado de *threads* não melhora a média de tempo de espera das *works units*, ou seja, o uso em grande quantidade de *threads* não garante uma boa performance do motor de execução.

REFERÊNCIAS BIBLIOGRÁFICAS

AALST, W. M. V. D. Business process simulation survival guide. In: Handbook on Business Process Management 1. [S.l.]: Springer, p. 337–370, 2015.

AALST, W. Van der; STAHL, C. Modeling business processes: a petri net-oriented approach. [S.l.]: MIT press, 2011.

ALKHANAK, E. N. et al. Cost optimization approaches for scientific workflow scheduling in cloud and grid computing: A review, classifications, and open issues. Journal of Systems and Software, Elsevier, v. 113, p. 1–26, 2016.

BLYTHE, J. et al. Task scheduling strategies for workflow-based applications in grids. In: IEEE. CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005. [S.l.], v. 2, p. 759–767, 2005.

DOSSOT, D.; D'EMIC, J.; ROMERO, V. Mule in action. [S.l.]: Manning Publications Co., 2014.

FRANTZ, R. Z., QUINTERO, A. M. R. e CORCHUELO, R. A domain-specific language to design enterprise application integration solutions. International Journal of Cooperative Information Systems, 20(02):143–176, 2011.

FRANTZ, R. Z., CORCHUELO, R. e ROOS-FRANTZ, F. On the design of a maintainable software development it to implement integration solutions. Journal of Systems and Software, 111:89–104, 2016.

FRANTZ, R. Z., CORCHUELO, R., BASTO-FERNANDES, V., ROSA-SEQUEIRA, F., ROOS-FRANTZ, F. e ARJONA, J. L. A cloud-based integration platform for enterprise application integration: a model-driven engineering approach. Software - Practice and Experience, 1:1–25, 2020.

FISHER, M. et al. Spring integration in action. [S.l.]: Manning Publications Co., 2012.

GEHLOT, V. e NIGRO, C. An introduction to systems modeling and simulation with colored petri nets. Em Proceedings of the 2010 winter simulation conference, páginas 104–118. IEEE, 2010.

HOHPE, G.; WOOLF, B. Enterprise integration patterns: Designing, building, and deploying messaging solutions. [S.l.]: Addison-Wesley Professional, 2004.

IBSEN, C.; ANSTEY, J. Camel in action. [S.l.]: Manning Publications Co., 2018.

JENSEN, K., KRISTENSEN, L. M. e WELLS, L. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. International Journal on Software Tools for Technology Transfer, 9(3):213–254, 2007.

LAW, A. M.; KELTON, W. D. Simulation modeling and analysis, mcgraw-hill. New York 2nd edition is also OK, v. 2, 2000.

MANIKAS, K. Revisiting software ecosystems research: A longitudinal literature study. Journal of Systems and Software, 117:84–103, 2016.

MURATA, T. Petri nets: Properties, analysis and applications. Proceedings of the IEEE, IEEE, v. 77, n. 4, p. 541–580, 1989.

PETERSON, J. L. Petri nets. ACM Computing Surveys (CSUR), ACM New York, NY, USA, v. 9, n. 3, p. 223–252, 1977.

PETRI, C. A. Kommunikation mit automaten. 1962.

SURHONE, L.; TIMPLEDON, M.; MARSEKEN, S. Petals Enterprise Service Bus (Esb). [S.l.]: Betascript Publishing, 2010.