PGCOMP - Programa de Pós-Graduação em Ciência da Computação
Universidade Federal da Bahia (UFBA)
Av. Milton Santos, s/n - Ondina
Salvador, BA, Brasil, 40170-110

https://pgcomp.ufba.br
pgcomp@ufba.br

SISTEMA DE SISTEMAS (SOS) É COMPOSTO POR UM CONJUNTO DE SISTEMAS, QUE INTERAGEM ENTRE SI PARA UM OBJETIVO COMUM. ASSIM, ELE TENDE A SER MAIOR E MAIS COMPLEXO DO QUE OS SISTEMAS TRADICIONAIS. PARA ABORDAR A QUESTÃO DA COMPLEXIDADE INERENTE A ESTE TIPO DE SISTEMAS COMPLEXOS, ELES SÃO FREQUENTEMENTE MODELADOS USANDO A BUSINESS PROCESS MODELING AND NOTATION (BPMN), QUE É UMA NOTAÇÃO PADRÃO PARA MODELAGEM DE PROCESSOS DE NEGÓCIOS. O DIAGRAMA DE COREOGRAFIA, INTRODUZIDO NA VERSÃO BPMN 2.0, FORNECE CONCEITOS ADEQUADOS PARA REPRESENTAR AS INTERAÇÕES ENTRE OS SISTEMAS CONSTITUINTES DE UM SOS. NO ENTANTO, OS MODELOS CRIADOS COM ESSA NOTAÇÃO PODEM CONTER ERROS, ALGUNS DOS QUAIS PODEM SER DETECTADOS EM TEMPO DE DESIGN E OUTROS APENAS EM TEMPO DE EXECUÇÃO. OS ERROS DE SINTAXE SÃO FACILMENTE DETECTADOS COM O AUXÍLIO DE FERRAMENTAS DE MODELAGEM, NO ENTANTO, A AUSÊNCIA DE UMA SEMÂNTICA FORMAL PARA BPMN TORNA MAIS DIFÍCIL IDENTIFICAR ERROS DE TEMPO DE EXECUÇÃO, COMO POR EXEMPLO, DEADLOCKS, LIVELOCKS E OUTRAS PROPRIEDADES DE SEGURANÇA EM DIAGRAMAS DE COREOGRAFIA DE BPMN. ESSES ERROS SÃO DIFÍCEIS DE DETECTAR E PODEM LEVAR A UMA OPERAÇÃO INADEQUADA OU ATÉ MESMO AO TRAVAMENTO DO SISTEMA. NESSE CONTEXTO, UM MÉTODO PARA IDENTIFICAR ERROS DE TEMPO DE EXECUÇÃO CONSISTE EM TRADUZIR O DIAGRAMA BPMN EM UM MODELO FORMAL QUE PODE SER ANALISADO EM UM VERIFICADOR DE MODELO. ASSIM, APRESENTAMOS UMA ABORDAGEM PARA CONSTRUIR UM MODELO FORMAL PARA OS DIAGRAMAS DE COREOGRAFIA DE BPMN EM TERMOS DA LINGUAGEM FORMAL π-ADL, QUE É UMA LINGUAGEM PROJETADA PARA A ESPECIFICAÇÃO DE ARQUITETURAS DINÂMICAS, UMA CARACTERÍSTICA INTRÍNSECA DOS SOS. PORTANTO, DEFINIMOS O MAPEAMENTO DOS ELEMENTOS DO DIAGRAMA DE COREOGRAFIA PARA π-ADL, A FIM DE AUXILIAR SUA DESCRIÇÃO FORMAL EM π-ADL. TAIS MODELOS -ADL PERMITEM SUA VERIFICAÇÃO FORMAL POR MEIO DE UM VERIFICADOR DE MODELO ESPECÍFICO, POSSIBILITANDO ASSIM, A DETECÇÃO PRÉVIA DE ERROS EM TEMPO DE EXECUÇÃO NO SISTEMA MODELADO.

Palavras-chave: NOTAÇÃO BPMN, BUSINESS PROCESS MODELING NOTATION (BPMN), SISTEMA DE SISTEMAS (SOS), LINGUAGEM π-ADL, COREOGRAFIA.

MSC | 119 | 2021

VERIFICAÇÃO FORMAL DE SISTEMAS DE SISTEMAS (SOS) MODELADOS EM DIAGRAMAS DE COREOGRAFIA (BPMN) PARA DETECÇÃO PRÉVIA DE ERROS TÍPICOS DE TEMPO DE EXECUÇÃO.

Leila de Carvalho Costa

UFBA

VERIFICAÇÃO FORMAL DE SISTEMAS DE SISTEMAS (SOS) MODELADOS EM DIAGRAMAS DE COREOGRAFIA (BPMN) PARA DETECÇÃO PRÉVIA

Leila de Carvalho Costa

Dissertação de Mestrado

Universidade Federal da Bahia

Programa de Pós-Graduação em
Ciência da Computação

Setembro | 2021

Universidade Federal da Bahia
Instituto de Computação

Programa de Pós-Graduação em Ciência da Computação

VERIFICAÇÃO FORMAL DE SISTEMAS DE SISTEMAS (SOS) MODELADOS EM DIAGRAMAS DE COREOGRAFIA (BPMN) PARA DETECÇÃO PRÉVIA DE ERROS TÍPICOS DE TEMPO DE EXECUÇÃO.

Leila de Carvalho Costa

DISSERTAÇÃO DE MESTRADO

Salvador
14 de setembro de 2021

LEILA DE CARVALHO COSTA

VERIFICAÇÃO FORMAL DE SISTEMAS DE SISTEMAS (SOS) MODELADOS EM DIAGRAMAS DE COREOGRAFIA (BPMN) PARA DETECÇÃO PRÉVIA DE ERROS TÍPICOS DE TEMPO DE EXECUÇÃO.

Esta Dissertação de Mestrado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientadora: Profa. Dra. Rita Suzana Pitangueira Maciel
Co-orientador: Prof. Dr. Adolfo Almeida Duran

Salvador
14 de setembro de 2021

Ficha catalográfica elaborada pela Biblioteca Universitária de Ciências e Tecnologias Prof. Omar Catunda, SIBI – UFBA.

C837 Costa, Leila de Carvalho

Verificação formal de Sistemas de Sistemas (SoS) modelados em diagramas de coreografia (BPMN) para detecção prévia de erros típicos de tempo de execução / Leila de Carvalho Costa. – Salvador, 2021.

71f.

Orientadora: Profa. Dra. Rita Suzana Pitangueira Maciel
Coorientador: Prof. Dr. Adolfo Almeida Duran

Dissertação (Mestrado) – Universidade Federal da Bahia, Instituto de Computação, 2021.

1. Business Process Modeling Notation (BPMN). 2. Sistema de Sistemas (SoS). 3. Linguagem π-ADL. 4. Engenharia de software. I. Maciel, Rita Suzana Pitangueira. II. Duran, Adolfo Almeida. III. Universidade Federal da Bahia. IV. Título.

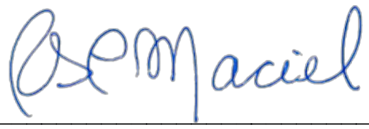
CDU:004.41

"*Uma Abordagem para Verificação de Erros de Tempo de Execução em SoS através de Diagramas de Coreografia BPMN*"

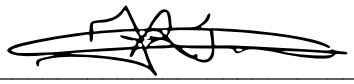
Leila de Carvalho Costa

Dissertação apresentada ao Colegiado do Programa de Pós-Graduação em Ciência da Computação na Universidade Federal da Bahia, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação.

Banca Examinadora



Profª. Drª. Rita Suzana Pitangueira Maciel(Orientadora-UFBA)



Prof. Dr. Flávio Oquendo (IRISA)

Profª. Drª. Lais do Nascimento Salvador (UFBA)

Aos meus pais, irms e meu namorado. Pelo carinho e apoio durante esta jornada, que tornou tudo mais fcil!

AGRADECIMENTOS

Agradeço primeiramente a Deus, pois mais uma vez Ele me fortaleceu para chegar ao fim desse desafio.

Agradeço à minha família pelo apoio que sempre me deram durante toda a minha vida, que serviu de alicerce para as minhas realizações. Aos meus pais, Tereza e Tadeu, por todo esforço investido em minha educação, pelo incentivo e por acreditaram em mim desde o primeiro instante. Agradeço as minhas irmãs Laiz e Layara, pela amizade e atenção dedicadas sempre que precisei e pela compreensão nos momentos de ausência dedicados ao estudo.

Agradeço a Magno, meu namorado, amigo e parceiro, que acompanhou de perto toda minha trajetória sempre me encorajando e fortalecendo nos momentos mais difíceis. Obrigada por toda dedicação oferecida, pelo seu companheirismo e pelas palavras de apoio em momentos necessários.

Aos meus orientadores Rita Suzana Pitangueira Maciel e Adolfo Almeida Duran, pela paciência, esforço e por durante estes anos me acompanharem pontualmente, dando todo o auxílio necessário para a elaboração deste trabalho. Agradeço aos professores Flávio Oquendo e Laís do Nascimento Salvador, por aceitarem o convite de participar da banca de defesa e ajudar a contribuir com a melhoria do nosso trabalho.

Meu agradecimento também a esta instituição por ter me proporcionado a estrutura necessária para que pudesse crescer academicamente e pessoalmente.

E, por fim, agradeço a todos que, de alguma forma, fizeram parte e indiretamente contribuíram para a realização deste trabalho.

Lembre-se que as pessoas podem tirar tudo de voc, menos o seu conhecimento.

—ALBERT EINSTEIN

RESUMO

Sistema de Sistemas (SoS) é composto por um conjunto de sistemas, que interagem entre si para um objetivo comum. Assim, ele tende a ser maior e mais complexo do que os sistemas tradicionais. Para abordar a questão da complexidade inerente a este tipo de sistemas complexos, eles são frequentemente modelados usando a Business Process Modeling and Notation (BPMN), que é uma notação padrão para modelagem de processos de negócios. O diagrama de coreografia, introduzido na versão BPMN 2.0, fornece conceitos adequados para representar as interações entre os sistemas constituintes de um SoS. No entanto, os modelos criados com essa notação podem conter erros, alguns dos quais podem ser detectados em tempo de design e outros apenas em tempo de execução. Os erros de sintaxe são facilmente detectados com o auxílio de ferramentas de modelagem, no entanto, a ausência de uma semântica formal para BPMN torna mais difícil identificar erros de tempo de execução, como por exemplo, deadlocks, livelocks e outras propriedades de segurança em diagramas de coreografia de BPMN. Esses erros são difíceis de detectar e podem levar a uma operação inadequada ou até mesmo ao travamento do sistema. Nesse contexto, um método para identificar erros de tempo de execução consiste em traduzir o diagrama BPMN em um modelo formal que pode ser analisado em um verificador de modelo. Assim, apresentamos uma abordagem para construir um modelo formal para os diagramas de coreografia de BPMN em termos da linguagem formal π-ADL, que é uma linguagem projetada para a especificação de arquiteturas dinâmicas, uma característica intrínseca dos SoS. Portanto, definimos o mapeamento dos elementos do diagrama de coreografia para π-ADL, a fim de auxiliar sua descrição formal em π-ADL. Tais modelos π-ADL permitem sua verificação formal por meio de um verificador de modelo específico, possibilitando assim, a detecção prévia de erros em tempo de execução no sistema modelado.

Palavras-chave: Notação BPMN, Business Process Modeling Notation (BPMN), Sistema de Sistemas (SoS), Linguagem π-ADL, Coreografia.

ABSTRACT

System of systems (SoS) is composed of a set of systems, which interact together for a common goal. Thereby, it tends to be larger and more complex than traditional systems. To address the question of the inherent complexity of this kind of complex systems, they are frequently modeled using the Business Process Modeling and Notation (BPMN), which is a standard modeling notation for business processes. The choreography diagram, introduced in version BPMN 2.0, provides suitable concepts for representing the interactions among constituents of a SoS. However, models created using this notation may contain errors, some of which can be detected at design-time and others only at runtime. Syntax errors are easily detected with the assistance of modeling tools. Nevertheless, the absence of a formal semantics for BPMN makes it harder to identify runtime errors, as for example, deadlocks, livelocks and other safety properties in BPMN choreography diagrams. These errors are challenging to detect and may lead to an improper operation or even a system lockup. In this context, a method for identifying runtime errors consists of translating the BPMN diagram into a formal model that can be analyzed in a model checker. Thereby we present an approach to build a formal model for the BPMN choreography diagrams in terms of the formal language π-ADL, which is properly designed for the specification of dynamic architectures, an intrinsic characteristic of SoS. Therefore, we define the mapping of the elements of the choreography diagram to π-ADL, in order to obtain its formal description in π-ADL. Such π-ADL models allow its formal verification using a specific model checker, enabling the prior detection of runtime errors in the modeled system.

Keywords: Business Process Modeling Notation (BPMN), System of Systems (SoS), π-ADL language, Choreography.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1

INTRODUÇÃO

Sistemas de Sistemas (SoS) são sistemas formados por outros sistemas heterogêneos e que operam de modo independente, mas que são interligados para um objetivo comum (JAMSHIDI, 2008) (BOARDMAN; SAUSER, 2006). Os sistemas que compõem um SoS são chamados de sistemas constituintes, onde a interação desses sistemas constituintes é composta possivelmente em tempo de execução, gerando sistemas maiores e mais complexos (KARCANIAS; HESSAMI, 2011). Uma das formas de lidar com a questão da complexidade inerente aos SoS é fazendo uso da modelagem de sistemas.

A BPMN (OBJECT MANAGEMENT GROUP, 2008) é uma notação muito utilizada para modelar fluxos de processos de negócio. Em SoS, BPMN é utilizada para coordenar e sincronizar processos de negócios automatizados, ajudar a descrever os sistemas, produtos, layout de serviços internos e a sua posição em relação uns aos outros, ou seja, a sua arquitetura (LUZEAUX; RUAULT, 2013).

BPMN fornece vários tipos de diagramas para planejamento e gerenciamento de processos de negócios, óticas e construções diferentes. O mais conhecido é o diagrama de orquestração, que possui seu foco em descrever o trabalho que é realizado pelos participantes do negócio. Já o diagrama focado nas interações entre os participantes do negócio através das suas trocas de mensagem é conhecido como diagrama de coreografia e foi apresentado apenas na versão 2.0 da notação.

Embora BPMN seja muito utilizada como ferramenta de modelagem de processos de negócios, os modelos criados usando essa notação podem conter erros indesejáveis. Eles podem ser erros que são normalmente encontrados analisando o modelo de processo usando algumas ferramentas de modelagem, mas também existem os erros que acontecem durante o tempo de execução do processo como os deadlocks e livelocks.

Esses erros de execução quando ocorrem, ocasionam o funcionamento inadequado ou até mesmo o travamento do sistema e são difíceis de serem detectados por ocorrer no momento em que o sistema está sendo executado. Em SoS a questão se potencializa, pois, neste cenário, um processo pode ser composto por um ou vários processos distintos, oriundos de diversos sistemas, tornando-se grande e complexo demais para que seja possível descobrir qual a origem do erro sem um suporte automatizado.

Na literatura são encontradas diferentes abordagens formais para detectar erros que ocorrem em tempo de execução em sistemas modelados em BPMN, como pode ser observado em (KHERBOUCHE; AHMAD; BASSON, 2012) e (KHERBOUCHE; AHMAD; BASSON, 2013) onde é proposto abordagens para automatizar a verificação de erros com base na verificação do modelo para a estrutura Kripke, que é uma variação do autômato não determinístico utilizado no modelo de verificação e representa o comportamento de um sistema. Em (AWAD; PUHLMANN, 2008) e (LAUE; AWAD, 2011), abordam a apresentação de erros de uma forma visual propondo detectar bloqueios em consultas gráficas dada em BPMN-Q, uma linguagem visual que é baseada na BPMN usada para consultar modelos de processos de negócios com base em sua estrutura e no trabalho (TANTITHARANUKUL; SUGUNNASIL; JUMPAMULE, 2010) apresentam uma abordagem de autômatos de estados finitos para verificar o modelo em várias terminações. A estratégia deste artigo é o de transformar o modelo BPMN para um formalismo baseado em autômatos de processo para verificar a compatibilidade da função de transição.

Detectar estes erros em sistemas modelados usando a notação BPMN não é um assunto novo, mas é uma questão relevante quando se trata da modelagem utilizando o diagrama de coreografia, pois as abordagens encontradas na literatura fazem referência a BPMN utilizando apenas diagrama de orquestração. Outra questão que esses trabalhos não atendem, é quando se trata de modelagem de Sistemas de Sistemas, visto que estes trabalhos possuem o foco apenas em modelagem de sistemas tradicionais.

Nosso trabalho visa a utilização do diagrama de coreografia, que é um diagrama proposto na versão BPMN 2.0 (OBJECT MANAGEMENT GROUP, 2011). Este diagrama representa mais fielmente a dinâmica da interação dos componentes de um SoS. A coreografia difere em comportamento e propósito de um processo padrão BPMN, já que trata a forma como os participantes do negócio coordenam suas interações. Seu foco não está em como o processo é feito e sim na troca de informações, através de mensagens entre esses participantes (OBJECT MANAGEMENT GROUP, 2011).

O objetivo deste trabalho é a construção de um modelo formal para o diagrama de coreografia BPMN, que deve permitir especificar o comportamento de um sistema modelado utilizando tal notação, possibilitando assim a verificação desse modelo formal em um verificador específico. Desse modo, buscamos avaliar se esse modelo formal nos permite revelar erros que ocorrem tipicamente em tempo de execução.

Para sua construção escolhemos trabalhar com a linguagem π-ADL (OQUENDO, 2004), pois já são encontradas abordagens correlatas que mostram como diagramas padrão BPMN podem ser descritos nessa linguagem formal. Além disso, ela é uma linguagem formal especialmente projetada para a especificação de arquiteturas dinâmicas (característica dos SoS) e também possui um conjunto de ferramentas que auxilia no processo de especificação e verificação.

Mesmo o nosso trabalho focando na utilização no diagrama de coreografia da BPMN 2.0, esses trabalhos serviram de embasamento para a definição etapas do nosso trabalho, como o mapeamento formal de elementos BPMN para π-ADL e a especificação da arquitetura dos modelos, aplicando uma composição dos fragmentos do mapeamento realizado.

Para tal, a partir desse mapeamento, traduzimos os elementos que compõe a notação BPMN em termos de π- ADL, obtendo assim, o modelo formal correspondente. Este

modelo formal deve permitir capturar o comportamento do sistema, viabilizando sua verificação em um verificador específico para linguagem π-ADL.

A metodologia de pesquisa utilizado para realizar a validação da abordagem proposta foi realizado seguindo basicamente as três etapas apresentadas a seguir:

- Revisão da literatura: busca a identificação das abordagens existentes sobre detecção de erros estruturais em modelos em BPMN disponíveis na literatura, além de auxiliar na decisão e conhecimento da linguagem formal utilizada nesse trabalho, e também na definição das etapas para a validação da abordagem proposta.
- Etapa de descrição: objetiva descrever formalmente, nos termos da linguagem formal π-ADL, cenários de sistemas de sistemas modelados utilizando diagramas de coreografia BPMN.
- Etapa de verificação: pretende verificar, com o auxílio de um modelo de verificação, se a formalização dos cenários na linguagem π-ADL permite detectar os possíveis deadlocks e livelocks que possam ocorrem durante sua execução, para isso utilizamos dois diagramas uma para exemplificação e outro mais complexo para a aplicação da proposta.

O restante do trabalho está organizado da seguinte forma. O Capítulo 2 apresenta aos principais aspectos sobre Sistema de sistemas (SoS), a notação BPMN, sobre a linguagem π-ADL e sobre verificação formal de modelos, que são conceitos muito importantes para o entendimento deste trabalho. No Capítulo 3, são expostos os trabalhos resultantes da revisão da literatura e que serviram de embasamento para a realização desse trabalho. No Capítulo 4, é apresentada a solução proposta no nosso trabalho. O Capítulo 5 apresenta a exemplificação da proposta e a análise dos resultados. Por fim, o Capítulo 6 trata das considerações finais, onde discutimos as principais contribuições do nosso trabalho e apresentamos sugestões para trabalhos futuros.

Capítulo 2

REFERENCIAL TEÓRICO

Neste capítulo, buscamos elaborar a contextualização da pesquisa e seu embasamento teórico, trazendo uma sistematização do conhecimento científico sobre os temas relevantes para o nosso trabalho. Aqui também são explicitados os principais conceitos e termos técnicos a serem utilizados na pesquisa. São eles: Sistemas de Sistemas (SoS), que é apresentado na Seção 2.1; A Notação de Modelagem de Processos de Negócios (BPMN) com ênfase no diagrama de coreografia, que é apresentado na Seção 2.2, a Seção 2.3 é dedicada a linguagem formal π-ADL e, por fim, a Seção 2.4 dedicada a Verificação de modelos que é a abordagem utilizada para a solução do problema proposta.

2.1 SISTEMAS DE SISTEMAS

O termo Sistema de Sistemas (SoS) tornou-se uma expressão bastante comum para classificar um conjunto de sistemas que interagem entre si (BALDWIN; SAUSER, 2009), onde os sistemas são selecionados e compostos possivelmente em tempo de execução para formar um sistema mais complexo (KARCANIAS; HESSAMI, 2011). Essa interação de sistemas tradicionais para compor um Sistema de Sistemas, forma um sistema mais eficiente, uma vez que eles tendem a ser maiores, mais complexos e operam em contextos mais variados que os sistemas tradicionais (MEKDECI et al., 2011).

Esse tipo de sistema é composto por outros sistemas diferentes e independentes, chamados sistemas constituintes, que são interligados para um propósito e finalidade em comum, podendo ser custo, desempenho, robustez, entre outros (JAMSHIDI, 2008) (BOARDMAN; SAUSER, 2006). Os sistemas componentes conseguem sozinhos objetivos bem fundamentados mesmo que sejam separados do sistema geral. No entanto, eles também funcionam para resolver os objetivos do conjunto, que surgem com o SoS, que geralmente não podem ser alcançados pelos sistemas de forma individual (SAGE; CUPPAN, 2001).

Como um SoS também é composto de partes organizadas para realizar uma função específica, está de acordo com a definição de sistemas tradicionais, logo um SoS é um sistema, mas nem todo sistema é um SoS. (XIAO et al., 2011).

Embora muitos Sistemas de Sistemas sejam distribuídos geograficamente, não é a complexidade ou o tamanho dos sistemas componentes ou sua distribuição geográfica que

os torna um "sistema de sistemas", mas a distribuição geográfica também pode ser vista como uma característica desses sistemas. Outra que auxilia para distinguir um Sistema de Sistemas de um sistema grande e complexo “comum”, é seu desenvolvimento evolutivo, ou seja, funções e propósitos são adicionados, removidos e modificados com o sistema em uso. Com essa característica evolutiva o sistema desenvolve a um comportamento emergente pelo qual ele funciona e desenvolve objetivos que não são possíveis por nenhum dos sistemas componentes individualmente (SAGE; CUPPAN, 2001).

Um SoS pode ser definido por cinco características (MAIER, 1996) que também estão presentes em muitos dos sistemas tradicionais, mas para que um sistema possa ser chamado de SoS é preciso que todas ou a maioria dessas cinco características estejam presentes (SAGE; CUPPAN, 2001). Um resumo destas características seria o seguinte:

- **Independência operacional:** um sistema de sistemas é composto por sistemas independentes e quando desfeito estes sistemas são capazes de executar operações úteis sem depender uns dos outros.
- **Independência gerencial:** os sistemas componentes mantém uma existência operacional contínua que é independente do SoS.
- **Distribuição geográfica:** os sistemas componentes geralmente tem uma grande dispersão geográfica.
- **Desenvolvimento evolutivo:** um sistema de sistemas nunca é totalmente formado ou completo, funções e propósitos são adicionados, removidos e modificados ao longo do tempo que ele é executado.
- **Comportamento emergente:** o sistema de sistemas executa funções que não estão em nenhum sistema componente, pois são propriedades emergentes que pertence ao SoS como um todo.

2.2 MODELAGEM DE PROCESSO DE NEGÓCIO

Modelagem de processo de negócio é o conjunto de atividades que compreende a criação e representação de processos de negócio existentes ou propostos. Tem como proposito criar uma representação do processo de maneira completa e precisa sobre seu funcionamento. Os processos de negócios podem ser representados através de um modelo em diversos níveis de detalhes, que vai de uma visão abstrata até uma visão mais detalhada, representando diversas perspectivas, servindo a diferentes propósitos (CBOK, 2013).

Um modelo de processos retrata os principais elementos de um fluxo de processo, representa as atividades, eventos, condições e outros elementos referentes ao processo. Geralmente esse modelo é criado apoiado em alguma notação de modelagem de processo, que é um conjunto de símbolos padronizados e as regras que determinam o significado desses símbolos, que ajudam a mostrar os relacionamentos que ocorrem durante o processo de negócio. A utilização de uma notação é importante, pois emprego de uma abordagem que segue normas conhecidas oferecem vantagens como (CBOK, 2013):

- Conjuntos de símbolos, técnicas e linguagens que facilitam a comunicação entre os envolvidos;
- Consistência dos modelos de processos resultantes;
- Possibilidade de importar e exportar os modelos para diferentes fermentas;
- Além da geração de aplicações a partir destes modelos.

A notação utilizada na realização desse trabalho é a Notação de Modelagem de Processo de Negócio (BPMN em inglês) (OBJECT MANAGEMENT GROUP, 2011), por apresentar um conjunto robusto de símbolos para modelagem de diferentes aspectos de processos de negócio, além de possuir o uso e entendimento difundido em muitas organizações e possuir uma versatilidade para modelar diversas situações de um processo (CBOK, 2013).

2.2.1 Noatação BPMN

A Notação de Modelagem de Processos de Negócios (BPMN) é uma notação gráfica adotada pela OMG (OBJECT MANAGEMENT GROUP, 2008) para modelar processos de negócio. O principal objetivo é fornecer às empresas a capacidade de compreender seus procedimentos de negócios internos em uma notação gráfica e a capacidade de comunicar esses procedimentos de uma maneira padrão. A BPMN segue a tradição de notações de fluxograma para facilitar a sua criação e leitura, fornecendo vários diagramas, que são projetados para uso das pessoas que projetam e gerenciam os processos de negócios.

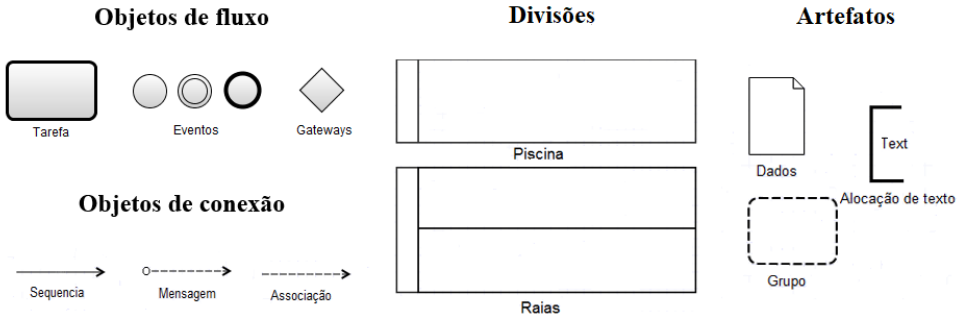


Figura 2.1 Elementos BPMN.

Seu desenvolvimento teve como propósito criar um método simples para gerar modelos de processo de negócios que pudessem capturar a complexidade inerente a esses processos. Para isso os aspectos gráficos da notação foram organizados em grupos específicos gerando um pequeno conjunto de categorias de notação onde facilmente se consegue identificar os tipos básicos de elementos e entender o diagrama (OBJECT MANAGEMENT GROUP, 2008). As quatro categorias básicas para a criação desses diagramas são apresentadas na Figura 2.1 e são classificadas da seguinte forma:

- **Objetos de fluxo:** São os elementos gráficos para definir o comportamento dos processos. Existem três tipos: eventos, tarefas e gateways.

- **Objetos de conexão:** São os elementos gráficos para conectar os objetos de fluxo entre si ou outras informações. Existem três tipos: fluxo de sequência, fluxo de mensagens e associação.
- **Divisões:** São elementos gráficos para agrupar e organizar os processos de modelagem. Tem uma estrutura semelhante a uma piscina e suas raias.
- **Artefatos:** São elementos gráficos usados para fornecer informações adicionais sobre os processos. Existem três tipos: objeto de dados, anotação e grupo.

Em um processo de negócio, algumas *tarefas* devem ser executadas ao longo do processo, talvez sob certas condições, apresentadas pelos *gateways*, e as coisa podem acontecer, exemplificadas com os *eventos*. O que conecta esses três *objetos de fluxos* são os *fluxos de sequência*, mas apenas dentro de uma piscina. Se as conexões cruzarem os limites da piscina, o processo recorre aos *fluxos de mensagem* (FREUND; RÜCKER, 2012).

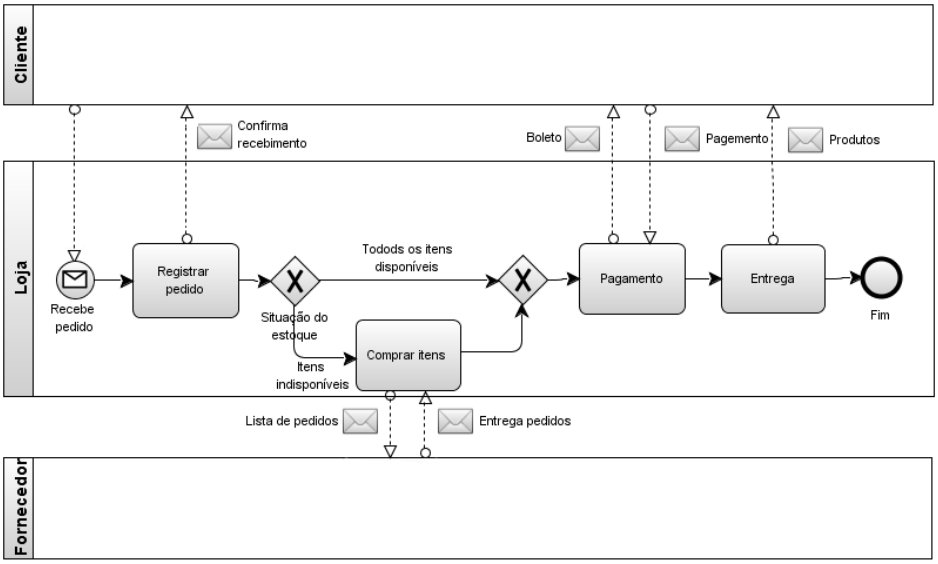


Figura 2.2 Exemplo de diagrama de orquestração.

BPMN fornece vários diagramas projetados para aqueles que planejam e gerenciam processos de negócios. Em um processo da realização de compra online apresentado na Figura 2.2, mostramos a forma de utilização dos elementos apresentados para a criação de um diagrama padrão, que é mais comum para a maioria dos modeladores de processos, pois está presente na notação BPMN desde a primeira versão. Esse diagrama é também conhecido como diagrama de orquestração e está focado em descrever o fluxo de atividades de uma entidade ou organização, isso é, a descrição do trabalho realizado pelos participantes do negócio.

A versão 2.0 (OBJECT MANAGEMENT GROUP, 2011) da notação BPMN ampliou o escopo e os recursos da notação em várias áreas como, por exemplo, oferecendo um novo modelo que é o modelo de coreografia, modelo no qual será baseado o nosso trabalho. O

Modelo de coreografia é um diagrama que difere em maneira de compreender diagrama de coreografias e como eles são usados em BPMN.

Ele difere em forma, propósito e comportamento do diagrama de orquestração, que, como já foi citado, tem o foco na descrição do trabalho realizado pelos participantes de uma organização. O diagrama de coreografia foca nas interações dos participantes do negócio, ou seja, nas suas trocas de informações que são realizadas através de mensagens. Embora sejam diagramas para expressar comportamentos, eles possuem óticas diferentes e, consequentemente, suas construções também diferem.

Para uma melhor compreensão do novo diagrama, apresentamos um diagrama de coreografia derivado do diagrama de orquestração apresentado na Figura 2.2. O diagrama de orquestração descreve as interações entre dois ou mais participantes de negócios, essas interações são definidas como uma sequência de atividades e também representa os padrões de troca de mensagens entre os participantes envolvidos, chamados de coreografia. As representações gráficas dos participantes são através de piscinas e a interação entre elas são fluxos de mensagens.

Porém um diagrama de coreografia é, como já foi dito, um tipo diferente de processo, pois define a sequência de interações entre participantes e fornece uma definição mais concisa porque não há piscinas, já que essas interações não são funções de um único participante, pelo contrário, cada estágio da coreografia envolve dois ou mais participantes e as interações são encapsuladas em um único objeto, uma Tarefa de Coreografia, conforme ilustrado na Figura 2.3. Com isso podemos afirmar que coreografia, em termos de BPMN, é estabelecida fora de qualquer piscina.

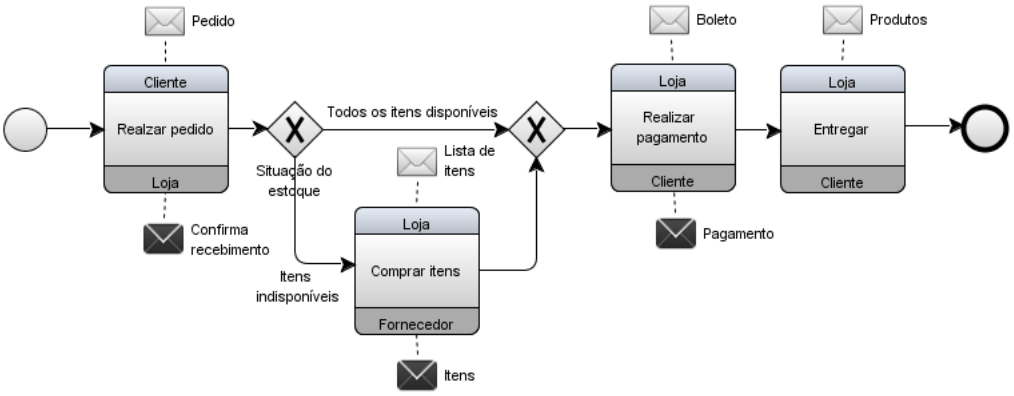


Figura 2.3 O mesmo processo da Figura 2.2 modelado em diagrama e coreografia.

A tarefa de coreografia é um novo elemento introduzido pelo diagrama de coreografia. Este elemento consiste na representação de uma ou duas trocas de mensagens entre pelo menos dois participantes reunidos em um único objeto. O nome da tarefa e de cada participante são apresentados nas diferentes faixas que formam sua notação gráfica, conforme mostrado na Figura 2.4. A faixa não sombreada corresponde ao participante que iniciou a interação.

Basicamente, cada participante só pode entender o status de uma coreografia através do comportamento observável de outro participante, que são as mensagens que foram

enviadas e recebidas. Se houver apenas dois participantes na coreografia, então é muito simples, ambos participantes serão cientes de quem é a responsabilidade para enviar a próxima mensagem. Entretanto, se houver mais que dois participantes, então o modelador precisa ser cuidadoso para sequenciar as atividades de coreografia de forma que os participantes saibam quando são responsáveis por iniciar as interações.

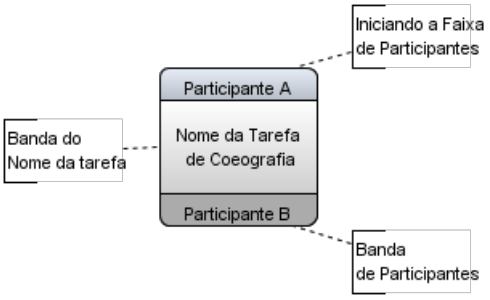


Figura 2.4 Representações de tarefas de coreografia.

A Figura 2.5 apresenta uma amostra de um processo modelado em um diagrama coreografia, onde o paciente busca auxílio médico para obter medicação para tratar sua doença. O diagrama apresenta em sua composição dois tipos de eventos, um inicial e um final, quatro atividades de coreografia e também cinco fluxos de sequência.

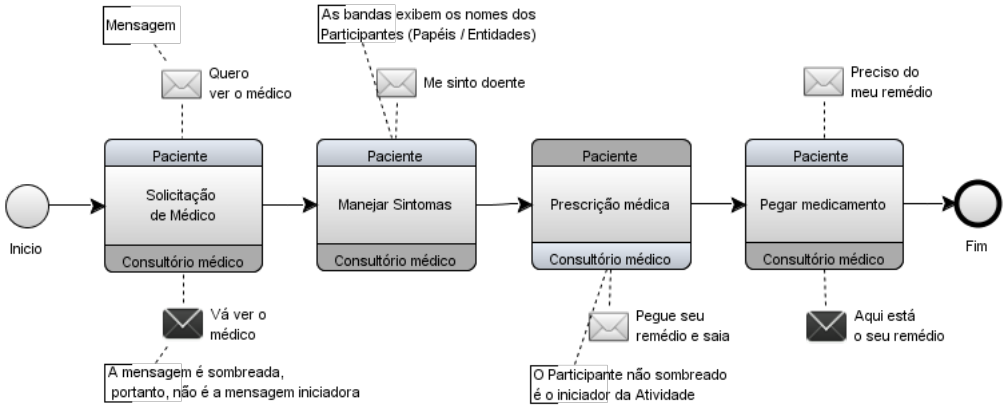


Figura 2.5 Exemplo de um diagrama de coreografia.

A seguir apresentaremos a descrição dos elementos que são comuns a todos os diagramas, alguns apresentados no diagrama de coreografia acima, juntamente com a descrição dos gateways, que não estão representados na figura, mas são de muita importância na construção de vários desses diagramas.

- **Fluxos de sequência:** Assim como no diagrama de orquestração, os fluxos de sequência são usados dentro das coreografias para mostrar a sequência das atividades de coreografias e só podem se conectar a outros objetos de fluxo. Em orquestração, eles se conectam a eventos, gateways e tarefas e em coreografias, eles se conectam a eventos, gateways e tarefas de coreografias, veja a Figura 2.6.

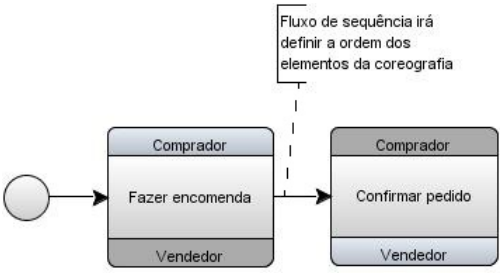


Figura 2.6 Representação do fluxo de sequência.

- **Eventos:** São usados da mesma maneira que se aplica no diagrama de orquestração. Existem três tipos principais, ver Figura 2.7, que são:

Figura 2.7 Representações de eventos.

  - **Evento de inicio:** E um marcador gráfico que indica onde inicia a coreografia. Podem ser do tipo simples, de tempo, condicional, entre outros.
  - **Eventos intermediários:** É um marcador gráfico que indica onde algo acontece em algum lugar entre o início e o fim da coreografia, eles são usados com pouca frequecia. Podem ser do tipo simples, de tempo, de link, entre outros.
  - **Eventos de final:** É um marcador gráfico para indicar onde termina um caminho dentro da coreografia. Podem ser do tipo simples ou terminação apenas.

- **Gateways:** Na coreografia os gateways são utilizados para criar caminhos alternativos e/ou paralelos, assim como em orquestração. Sendo assim, as interações entre participantes podem acontecer em sequência, em paralelo, ou através de seleção exclusiva. Os tipos de gateways são apresentados na Figura 2.8 e descritos a seguir.

  - **Gateways exclusiva:** São usadas para criar caminhos alternativos dentro de um processo ou uma coreografia.

– **Gateways inclusiva:** São usadas para modelar pontos de sincronização de um número de ramos.

– **Gateways paralela:** São usadas para criar caminhos e são executados ao mesmo tempo, dentro de um fluxo de coreografia.

Figura 2.8 Representação de passagens.

2.3 LINGUAGEM DE DESCRIÇÃO FORMAL Π-ADL

π-Architecture Description Language (π-ADL) (OQUENDO, 2004) é uma linguagem formal baseada na linguagem π-Calculus e tem sido utilizada para a descrição de arquitetura. E uma linguagem de especificação formal que foi projetada pelo *ArchWare European Project* e que nos permite especificar arquiteturas estáticas, móveis e também dinâmicas, característica muito importante quando se trabalha com SOS, visto que são sistemas de aspecto dinâmico, pois são compostos em tempo de execução.

Com π-Calculus é também possível se fazer especificação formal de comportamento em tempo de execução, mas não é adequado como uma Linguagem de Descrição Arquitetural (ADL), pois não fornece construções que permitam aos arquitetos expressar arquiteturas convenientemente. Já π-ADL é mais adequada porque combina a possibilidade de uma alta expressividade da arquitetura com uma notação formal bem definida. Resolvemos trabalhar com essa linguagem formal devido ao fato dela possuir um conjunto de ferramentas de especificação e para verificação automatizada das propriedades dos modelos.

Uma arquitetura é descrita em π-ADL levando em consideração seus componentes, conectores e a composição entre eles. Os componentes representam os elementos funcionais do sistema e são descritos em termos de portas externas e um comportamento interno e conectores são descritos nos mesmos termos, porém seu papel arquitetônico é conectar componentes, ou seja, gerenciam interações entre os componentes.

2.3.1 Sistema de tipos

A linguagem π-ADL é formalmente definida por um sistema formal de transição e tipo (OQUENDO, 2004). E esse sistema formal é definido em uma abordagem estruturada em três camadas principais.

- π-ADL_B, camada de base, que fornece construções de conexões e comportamento;
- π-ADL_{FO}, camada de primeira ordem, que estende o π-ADL_B com os tipos de dados base e construtores;

- π-ADL_{HO}, sistema formal da camada de ordem superior, fornece π-ADL com cidadania completa.

Em π-ADL_{FO} temos definidos os tipos de dados atômicos e compostos e π-ADL_B temos definidos os tipos de comportamento (CAVALCANTE; OQUENDO; BATISTA, 2014b). Esses tipos que serão apresentados nas seções que seguem.

2.3.1.1 Tipos básicos Os tipos de valor básico são utilizados para representar valores atômicos e esses valores são definidos na camada π-ADL_{FO} e são apresentados na Tabela 2.1

Tabela 2.1 Tipos básicos

Tipo	**Representação sintática**	**Definição**
Natural	`Natural`	Números naturais
Inteiro	`Integer`	Números inteiros
Real	`Real`	Números reais
Booleano	`Boolean`	Valores lógicos booleanos
Sting	`String`	Cadeia de caracteres

2.3.1.2 Tipo construídos Os tipos de valores construídos são baseados em construtores, fornecidos pela camada π-ADL_{FO}, para definir tipos compostos usando os tipos básicos. Esses valores são apresentados na Tabela 2.2 e suas representações exemplificadas no decorrer da seção.

Tabela 2.2 Tipos construídos

Tipo	**Representação sintática**	**Definição**
Tupla	`tuple` $[T_1, T_2, ..., T_n]$	Tupla $(v_1, v_2, ..., v_n)$ em que v_i tem um tipo T_i
Visão	`view` $[l_1 : T_1, l_2 : T_2, ..., l_n : T_n]$	Forma rotulada de uma tupla $(v_1, v_2, ..., v_n)$ em que v_i tem um rótulo l_i do tipo T_i
Localização	`location [T]`	Variável do tipo T em que o valores podem ser armazenados (escrever) e recuperados (ler)

Tupla: Como exemplo, considere a declaração:

```
t is tuple [Integer, String]
```

Refere-se à declaração de uma tupla t associada a pares em que o primeiro valor é um valor *inteiro* e o segundo é um valor de *string*.

Visão: Como exemplo, considere a declaração:

```
v is view [x :  Integer, y :  String]
```

Refere-se à declaração de uma visão v associada aos pares em que o primeiro valor x é um valor *inteiro* e o segundo y é um valor *string*.

Localização: Como exemplo, considere a instrução:

```
l is location[Integer]
```

Declara uma variável inteira nomeado como l. Valores armazenados em uma variável tal pode ser lido usando seu identificador l e os valores podem ser gravados nele usando expressões de atribuição muito semelhantes aos seus homólogos em linguagens de programação.

2.3.1.3 Tipo de coleção Construtores são fornecidos pela camada π-ADL_{FO}, para definir tipos de coleção usando os tipos de valor básico e construído. Esses valores são apresentados na Tabela 2.3 e suas representações exemplificadas no decorrer da seção.

Tabela 2.3 Tipos de coleção

Tipo	**Representação sintática**	**Definição**
Mapa	`map` $[T_1, T_2]$	Mapa de elementos associando uma chave do tipo T_1 com valores do tipo T_2
Conjunto	`set [T]`	Coleção não ordenada de elementos do tipo T sem duplicatas
Sequência	`sequence [T]`	Coleção ordenada de elementos do tipo T que podem ter duplicatas

Mapa: Como exemplo, considere a declaração:

```
m is map[String, Integer]
```

Refere-se à declaração de um mapa m cujas chaves são valores de *string* associados a valores *inteiros*.

Conjunto: Como exemplo, considere a declaração:

```
s is set[Integer]
```

Refere-se à declaração de um conjunto s de valores *inteiros*.

Sequência: Como exemplo, considere a declaração:

```
q is sequence[Integer]
```

Refere-se à declaração de uma sequência q de valores *inteiros*.

2.3.1.4 Tipo de comportamento Construções de comportamento são fornecidos pela camada π-ADL_B para representar o comportamento interno de elementos arquitetônicos, que fazem uso de conexões para enviar e receber valores. Na Tabela 2.4 são apresentados os tipos de comportamentos oferecidos por π-ADL e suas representações exemplificadas no decorrer da seção.

Tabela 2.4 Tipos de comportamento

Comportamento	**Representação Sintática**	**Ação**
Tipo	`type s is T`	Cria um pseudônimo s para o tipo T
Prefixo de entrada	`via c send v`	Envia valor v via conexão c
Prefixo de saída	`via c send s:T`	Recebe valor s do tipo T via conexão c
Prefixo de condição	`if b do p`	Decretar prefixo p se a condição b é verdadeira
Prefixo silencioso	`unobservable`	Ação interna
Comportamento de escolha	`choose` B_1 `or` B_2`...` `or` B_n	Escolha para executar o comportamento B_1 ou comportamento B_2... B_n
Comportamento de composição	`compose` B_1 `and` B_2`...` `and` B_n	Executar comportamentos B_1... B_n em paralelo

Tipo: Como exemplo, considere as seguintes instruções:

```
type inicio is Real

type mensagem is String
```

As instruções criam os tipos *inicio* e *mensagem* definidos por um valor do tipo *real* e *string*, respectivamente.

Prefixos: são os comportamentos que executam ações apresentada em prefixos como o prefixo de entrada, prefixo de saída, prefixo de condição e prefixo de inação, e todos se apresentam como está exemplificado na Tabela 2.4.

Comportamento de escolha: Como exemplo, considere as seguintes instruções:

```
choose {
  via input receive v :  Integer
  via output send (v+1)
} or {
  via alt receive s :  Integer
  db add [s]
```

}

Este comportamento é expresso por duas alternativas de escolha, onde: (i) recebe um valor do tipo inteiro v através da conexão de entrada *input* e, em seguida, envia o incremento de v por meio da conexão de saída *output*, ou (ii) recebe um valor do tipo inteiro s através da conexão *alt* e, em seguida, adicionando-o ao conjunto *dB*.

Comportamento de composição: Como exemplo, considere as seguintes instruções:

```
compose {
  via input receive v :  Integer
  via output send (v+1)
} and {
  via signal send true
}
```

Neste comportamento composição, um valor do tipo inteiro v é recebido via conexão de entrada *input* e o seu incremento é enviado via conexão de saída *output*, simultaneamente ao envio do valor de typo booleano *true* através da conexão de saída *signal*.

Inação: É representado pela palavra *done* e, como o nome já sugere, representa um comportamento que não executar qualquer ação.

2.3.2 Descrição formal

Para especificar uma arquitetura em π-ADL é preciso fazer a descrição de cada componente e conectores presente e uma discrição arquitetura que mostra justamente a interação desses componentes e conectores. Para unir um componente a um conector, pelo menos uma conexão do primeiro deve ser unificada a uma conexão do último, de modo que as conexões conectadas podem transportar valores, conexões ou até mesmo elementos arquitetônicos (OQUENDO, 2004).

Os componentes e conectores são descritos da mesma maneira diferenciando apenas pela palavra-chave que identifica o tipo de elemento, no caso *component* e *connector* respectivamente. Essas palavras-chave são acompanhadas de um nome identificador e uma lista parâmetros e podem apresentar uma sequência de instruções. Cada uma dessas instruções deve ser descrida conforme as regras apresentadas abaixo (CAVALCANTE; OQUENDO; BATISTA, 2014a):

Declarações de tipo: podem ser declarados na forma *type s is T*, onde s é um identificador e T é um tipo existente. Como exemplo, considere a declaração de tipo:

```
type Key is String
```

Cria um tipo chamado *Key* que possui tipo *string*.

Declarações de conexão: podem ser declaradas na forma *connection c é d (T)*, em que *c* é um identificador, *d* é a direção da conexão, e *T* é um tipo existente. Para declarar a direção duas instruções são possíveis, pode ser *in*, para conexões de entrada e *out*, para as conexões de saída. Como exemplo, considere as instruções:

```
connection x is in(String)

connection y is out(Boolean)
```

Dessa forma declara que a conexão de entrada *x* recebe um valor *String* e a de saída *y* envia um valor *Booleano*.

Declarações variáveis: podem ser declaradas na forma *s is location [T]*, onde *s* é um identificador e *T* um tipo existente. Como exemplo, considere a instrução:

```
a is location[Real]
```

Isso declara uma variável *a* de um tipo *Real*.

Declaração de protocolo: sua declaração é opcional. Ele é declarado como um conjunto de ações de conexão, de acordo com sua declaração, que especificam a ação a ser executada pela conexão (envio ou recebimento) e o tipo dos valores que serão transmitidos através da conexão. Como exemplo, considere a seguinte declaração:

```
protocol is {
  ((via a receive Integer | via b receive Integer) via c send Integer)*
}
```

Impõe a recepção de um valor *inteiro* via conexão *a* ou via conexão *b* e depois enviando um valor *inteiro* via conexão *c*. Um caractere asterisco (*) é usado para especificar que uma determinada ação é executada zero ou mais vezes, já o caractere mais (+) é usado para especificar que é executada pelo menos uma vez.

Declaração de comportamento: um comportamento deve ser declarado obrigatoriamente para especificar o comportamento do elemento arquitetural. Tal comportamento compreende um conjunto de instruções, conforme definido nas seções anteriores:

- Declaração de tipo;
- Declaração variável;
- Declaração de coleções;
- Declaração de função e chamada de função com respectivo identificador, parâmetros e tipo de retorno;

- Prefixos (entrada, saída, condicional ou silencioso);
- Comportamento de escolha;
- Comportamento de composição;
- Chamada de comportamento recursivo;
- Inação.

De forma similar aos componentes e conectores, a arquitetura é descrita com sua palavra-chave, nesse caso *architecture*, acompanhadas de um nome identificador e uma possível lista paramentos. No entanto, apresenta como instrução um comportamento, que contém dois elementos básicos que são um conjunto que declara as instâncias de elementos e um conjunto declara as unificações. As instâncias do elemento estão contidas em um comportamento de composição e são da forma i é E, em que i é o identificador da instância e E é o identificador de um elemento arquitetural (componente ou conector). Já as unificações são o meio de passar valores de uma conexão de saída de um elemento para uma conexão de entrada de outro.

2.4 VERIFICAÇÃO DE MODELOS

Quando abordamos o problema do surgimento de erros em tempo de execução dos sistemas, sabemos que é algo que é corriqueiro, pois a principal razão pela qual esses estes erros ocorrem é simples: esta é uma atividade humana e os humanos são propensos a erros. Um bom meio que temos para encontrar e corrigir esses erros é usar o poder computacional dos computadores contra eles. A verificação do modelo lógico é uma das técnicas que nos permite fazer isso. São métodos que tentam automatizar partes específicas do processo de design (HOLZMANN, 2002).

Estes métodos são baseados no uso de autômatos e lógica de uma forma ou de outra. Isso inclui análise estática e métodos de interpretação simbólica, analisadores de acessibilidade e ferramentas de verificação do modelo lógico. Tais ferramentas tentam superar as limitações dos testes tradicionais, construindo e analisando modelos fechados, ou representações simbólicas de artefatos do mundo real, buscando a capacidade de resolver problemas computacionalmente difíceis ou de encontrar aproximações razoáveis da solução (HOLZMANN, 2002).

Há uma grande vantagem em poder verificar a exatidão dos sistemas de computador, sejam eles hardware, software ou uma combinação deles, isso é ainda mais óbvio no caso de sistemas críticos. A vantagem de realizar uma verificação formal é devido ao fato desse método determinar com precisão se um sistema pode satisfazer as propriedades que se deseja investigar (HUTH; RYAN, 2004).

Métodos de verificação formal se tornaram cada vez mais utilizados e à medida que as técnicas foram sendo adotadas, alguns problemas foram descobertos, o mais citado foi a complexidade computacional, surgindo a discussão sobre o chamado problema de explosão espacial de estados. Embora muitos algoritmos de verificação tenham uma complexidade que seja apenas linear no número de estados alcançáveis do sistema, este número pode se

sujeitar ao valor de alguns parâmetros críticos do problema que incluem, por exemplo, o número de processos de execução (HOLZMANN, 2002).

Como forma de afastar o problema de explosão de estados, o método de verificação de modelo estatística (SMC - Statistical Model Checker) tem sido utilizado. Além de promover melhor escalabilidade e menor consumo computacional, ele também permite suportar especificação formal de arquitetura de sistemas dinâmicos.

2.4.1 Verificação estatística

O método de verificação de modelo estatística é uma técnica probabilística que usa de simulação amostradas aleatoriamente destinada a verificar as probabilidades de um modelo de satisfazer, durante a execução do sistema, as propriedades de requisitos fornecidas. Ao contrário as técnicas convencionais de verificação do modelo, a SMC não tem os problemas de explosão de estado, pois não analisa a lógica interna do sistema. Assim, é mais adequado para validar sistemas críticos e complexos (KIM et al., 2013).

Para realizar a verificação é preciso de um modelo estocástico executável para o sistema, e as propriedades, a serem verificadas, expressas em uma determinada linguagem. E então, o SMC executa uma série de simulações estocásticas do sistema e avalia a probabilidade aproximada do sistema para atender a propriedade sob verificação (LEGAY; DELAHAYE; BENSALEM, 2010).

Os elementos de entradas para um verificador de modelo estatístico, consiste em um simulador para executar o programa, um verificador de modelo de lógica temporal linear limitada (BLTL) para verificar propriedades e um analisador estatístico responsável pelo cálculo de probabilidades, realizando testes estatísticos. (KIM et al., 2013).

Logo, em uma visão geral, as técnicas SMC consistem na simulação de um modelo executável do sistema que se quer verificar, considerando um conjunto de fórmulas que expressam propriedades a serem verificadas.

2.4.2 PLASMA

O PLASMA é uma biblioteca de verificação de modelo estatístico (SMC) escrita em Java, que possui uma classe de simulador personalizável que permite a prototipação rápida de soluções de verificação formal. Existem várias outras ferramentas SMC, a maioria usando uma única linguagem de modelagem relacionada a uma semântica específica, mas PLASMA surgiu como uma maneira de habilitar a análise formal de semântica de modelagem múltipla em uma única plataforma (BOYER et al., 2013).

É fornecido como um arquivo "jar"pré-compilado e um modelo de origem para criar a classe de simulador. Pode ser executado a partir de linhas de comando ou incorporado em outro software como biblioteca. Dispõe também de uma interface gráfica de usuário que fornece um ambiente de desenvolvimento integrado que facilita sua utilização como uma aplicação independente, com várias linguagens de modelagem. Seu ambiente de desenvolvimento facilita a simulação distribuída, e funciona com vários plug-ins de linguagens definidas pelo usuário (BOYER et al., 2013).

Entre as ferramentas SMC disponíveis na literatura, o PLASMA é uma plataforma compacta que permite aos usuários criar plug-ins de linguagens SMC personalizados em

cima dele. Devido a isso, o PLASMA foi utilizado para servir como base para o desenvolvimento de uma cadeia de ferramentas para verificação de propriedades de arquitetura de software dinâmicos como é apresentado em (CAVALCANTE et al., 2016).

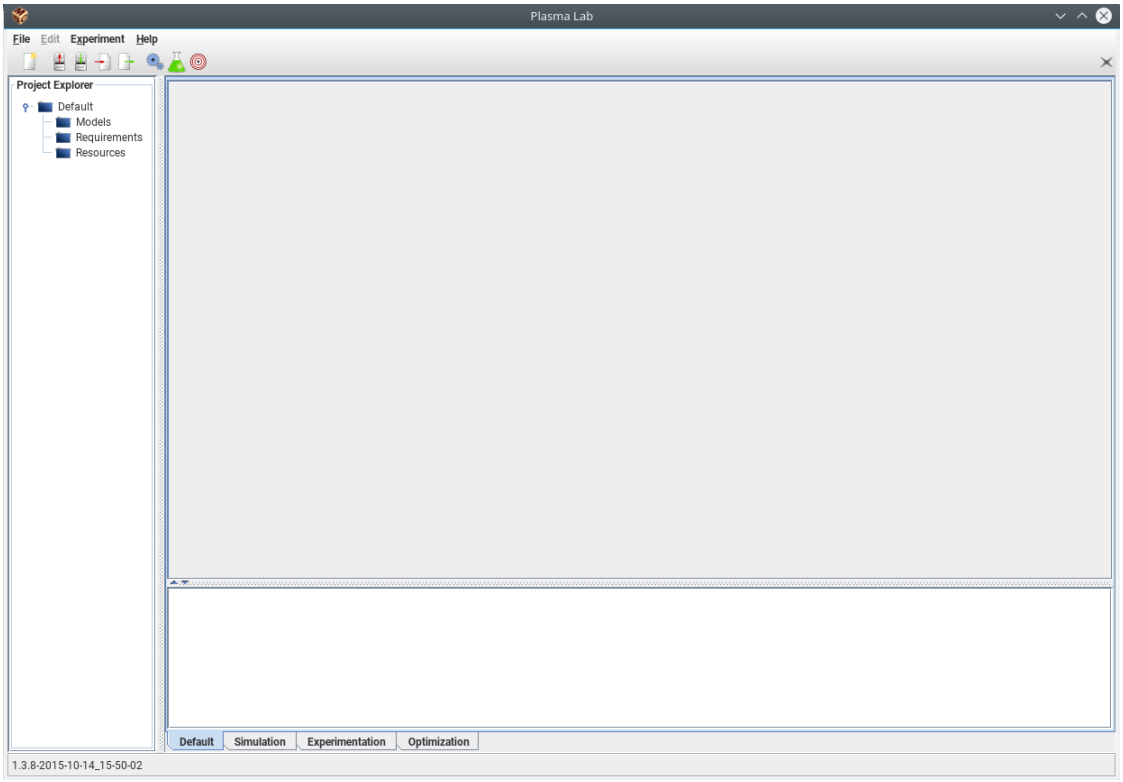


Figura 2.9 Janela inicial do PLASMA.

Sua interface gráfica pode ser observada na Figura 2.9, que é a visão que terá após inicializar a ferramenta.

Neste painel é onde podemos criar nossos projetos e onde inserimos os dados referentes ao modelo que descreve nosso sistema que e também as propriedades que queremos verificar. Outra tela que nos interessa é a de Experimentação, Figura 2.10, que é onde verificamos o nosso modelo.

Essa tela é composta de um painel de seleção de arquivo, um de algoritmo, um de execução e um de resultados. Para realizar a verificação selecionamos o projeto, o modelo do sistema bem como a propriedade que queremos verificar, no painel de seleção, escolhemos o algoritmo e definimos seus parâmetros, para, por fim, realizar a verificação.

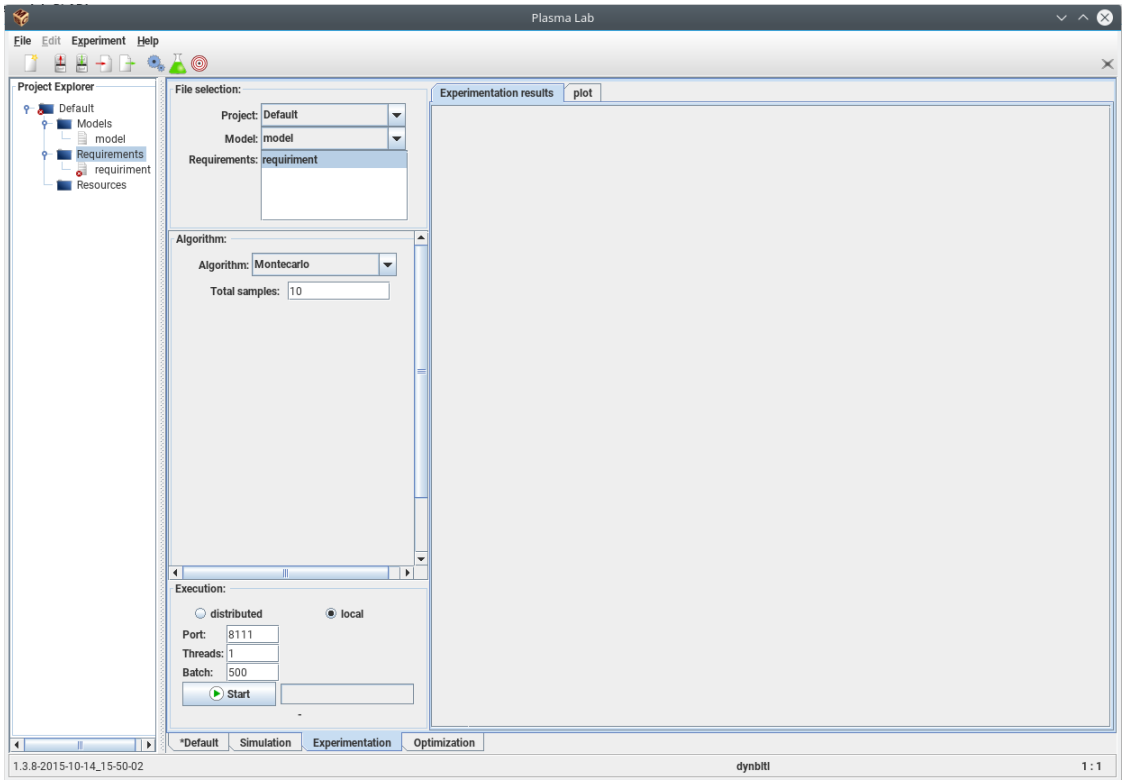


Figura 2.10 Tela de experimentação.

Capítulo

3

TRABALHOS RELACIONADOS

Neste capítulo, apresentamos uma visão geral das obras científicas disponíveis que tratam dos assuntos abordados neste trabalho, ou que estão sendo realizados no nosso campo de pesquisa.

Realizamos uma revisão ad-hoc da literatura, nas principais bases de pesquisa com o objetivo de identificar as principais técnicas para a identificação dos erros em tempo de execução em modelos BPMN, especialmente deadlocks e livelocks. Analisamos manualmente os proceedings das linhas de pesquisa relacionadas à área inicialmente de BPMN.

Esta pesquisa resultou em trabalhos focados na utilização de métodos formais para detecção destes erros, com base nisso definimos qual abordagem formal se adequava aos nossos objetivos e escopo, definindo a linguagem a ser utilizada obtendo outro foco de pesquisa, desta vez com foco na linguagem formal π-ADL, resultando em trabalhos que serviram de embasamento teórico para o desenvolvimento da elaboração da solução proposta, onde definimos as etapas realizadas para alcançar nossos objetivos.

A motivação do trabalho tem sido a de fornecer um meio para verificação de erros que ocorrem em tempo de execução em Sistemas de Sistemas modelados usando diagrama de coreografia BPMN. Detectar estes erros em sistemas modelados usando a notação BPMN não é um assunto novo, pois na literatura são encontradas diferentes abordagens formais que auxiliam sua identificação, mas é uma questão relevante quando se trata da modelagem utilizando o diagrama de coreografia, visto que as abordagens encontradas fazem referência a BPMN utilizando apenas diagrama de orquestração. Outra questão que esses trabalhos não atendem é quando se trata de modelagem de Sistemas de Sistemas, visto que estes trabalhos possuem o foco apenas em modelagem de sistemas tradicionais.

Este capítulo está organizado da seguinte forma: na Seção 3.1 discutimos brevemente alguns desses trabalhos existentes, onde foram encontrados métodos de detecção destes erros utilizando práticas variadas; e na Seção 3.2 abordamos a utilização de π-ADL em trabalhos que utilizam BPMN e em 3.3 apresentamos as etapas definidas com base nos trabalhos aqui apresentados.

3.1 DETECÇÃO DE LIVELOCKS E DEADLOCKS (ABORDAGENS UTILIZANDO VERIFICAÇÃO DE MODELOS)

Em (KHERBOUCHE; AHMAD; BASSON, 2012) e (KHERBOUCHE; AHMAD; BASSON, 2013) são propostas abordagens para automatizar a verificação de alguns erros em tempo de execução em modelos de processos BPMN, com base na verificação do modelo para a estrutura Kripke, que é uma variação do autômato não determinístico utilizado no modelo de verificação e representa o comportamento de um sistema.

Os autores em (AWAD; PUHLMANN, 2008) e (LAUE; AWAD, 2011) abordam a apresentação de erros de uma forma visual propondo detectar deadlocks em consultas gráficas dada em BPMN-Q, uma linguagem visual que é baseada na BPMN usada para consultar modelos de processos de negócios com base em sua estrutura. No trabalho (TANTITHARANUKUL; SUGUNNASIL; JUMPAMULE, 2010) apresentam uma abordagem de autômatos de estados finitos para verificar o modelo BPMN em várias terminações. A estratégia deste artigo é o de transformar o modelo BPMN para um formalismo baseado em autômatos de processo para verificar a compatibilidade da função de transição. Mas em nas três propostas acima citadas não há detecção de livelock.

Com base na análise dos trabalhos citados acima, pudemos observar de que forma são estruturados os trabalhos, os quais abordam a utilização de métodos formais para realizar verificação de modelos em sistemas modelados em BPMN. Onde se nota à similaridade dos trabalhos, que consistem em definir a tradução dos elementos da notação nos termos da linguagem escolhida, definir as propriedades utilizando a lógica que satisfaz a ferramenta utilizada para realizar a verificação do modelo.

Partindo disso, buscamos trabalhos que pudesse auxiliar na realização dessas etapas e buscando soluções usando linguagens formais que atendessem a nossa necessidade de especificar a arquitetura de SoS. Com isso, foram encontrados os trabalhos que estão apresentados na seção que segue.

3.2 TRABALHOS UTILIZANDO π-ADL

Como linguagem formal, escolhemos trabalhar com a linguagem π-ADL, porque além de serem encontradas abordagens que mostram como diagramas padrão BPMN, podem ser descritos na linguagem formal, conforme é apresentado em (OQUENDO, 2008a) e (OQUENDO, 2008b), ela é uma linguagem formal e foi projetada para a especificação de arquiteturas dinâmicas, característica dos SoS.

Em (OQUENDO, 2008a) o autor apresenta uma abordagem formal para o desenvolvimento de processos de negócios (modelados em BPMN) em termos de arquiteturas dinâmicas orientadas a serviços. Este trabalho traz um mapeamento formal de um conjunto de elementos do diagrama de orquestração BPMN em termos de π-ADL para gerar a especificação formal do modelo em π-ADL, porém esta especificação do processo não é apresentada no trabalho. Este mapeamento serviu de base para a criação do nosso mapeamento, mas com elementos do diagrama de coreografia.

Já em (OQUENDO, 2008b) é apresentado π-ADL para o desenvolvimento formal de composições dinâmicas de serviços da Web, a abordagem ilustra como BPMN pode ser

usado como uma notação visual para descrever os processos de negócios dessas composições e como eles são formalmente traduzidos usando π-ADL. Este trabalho mostra a aplicação do mapeamento presente em (OQUENDO, 2008a) para a especificação da arquitetura de um diagrama de orquestração apresentado.

Esses dois trabalhos serviram de embasamento para a definição dos passos da primeira etapa do nosso trabalho, que são justamente o mapeamento formal de elementos BPMN para π-ADL e a descrição da arquitetura dos modelos, aplicando uma composição dos fragmentos do mapeamento realizado. A diferença é que nossa abordagem é em relação a processos modelados usando o diagrama de coreografia e, em ambos os trabalhos citados, é apresentado essas aplicações no diagrama padrão BPMN (diagrama de orquestração).

Para auxiliar na etapa da verificação usamos o trabalho (CAVALCANTE, 2016). Onde é apresentado uma linha de ferramentas desenvolvida para verificar arquiteturas de software dinâmico descritas em π-ADL, tendo o verificador como base a ferramenta PLASMA que é uma ferramenta de verificação de modelo estatístico.

3.3 DEFINIÇÃO DAS ETAPAS DO ESTUDO

Com base em todos os trabalhos aqui citados conseguimos definir nossa proposta. Seguindo os trabalhos da seção 3.1 definimos que se trataria de uma abordagem formal, e juntamente com os trabalhos da seção 3.2 definimos os passos para a validação da nossa proposta, estas etapas são representadas na Figura 3.1 e explicadas na sequência:

Figura 3.1 Etapas de desenvolvimento do trabalho.

- A etapa de descrição que consiste em descrever nos termos da linguagem formal π-ADL um cenário modelado utilizando BPMN;
- A etapa de verificação que consiste em verificar, com o auxílio de um modelo de verificação (do inglês, Model Checking), se a formalização do cenário em BPMN na linguagem π-ADL ajuda na detecção de possíveis deadlocks e livelocks que possam ocorrem na execução do sistema modelado;
- E por fim a etapa de análise dos resultados que, como o nome já diz, através da averiguação dos resultados obtidos da verificação chegamos à conclusão desse trabalho.

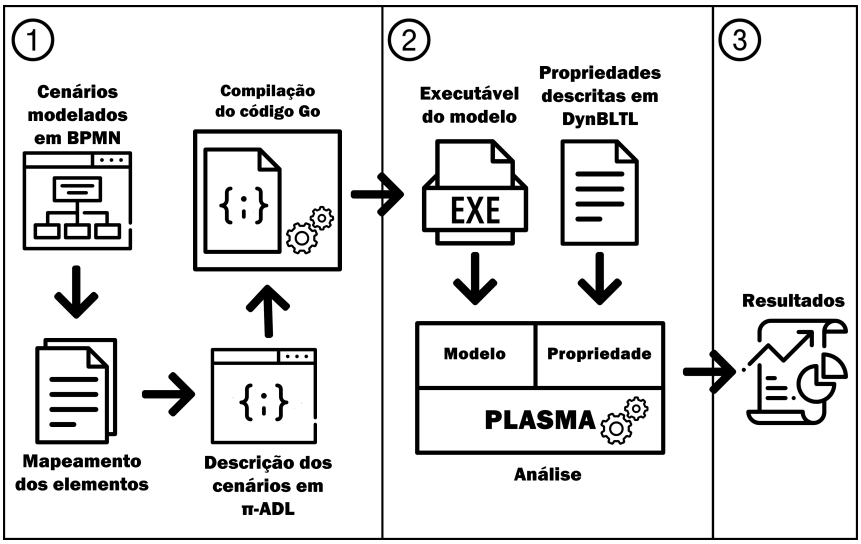


Figura 3.2 Etapas para realização do trabalho com seus passos detalhados.

Cada uma dessas etapas seguiu um conjunto de passos para alcançar resultados, como apresentado na Figura 3.2, onde:

- Na etapa de descrição são realizados:
  - Definição dos cenários modelados em BPMN a serem formalizados: Elaboração dos cenários no contexto de SOS modelados em diagramas de coreografia que apresentem deadlocks e livelocks conhecidos, para serem utilizado na experimentação. Nesse trabalho elaborados cenários em níveis de complexidade diferentes.
  - Mapeamento dos elementos BPMN para π-ADL: Consiste na descrição do comportamento de cada um dos elementos de modelagem do diagrama de coreografia em termos de π-ADL, para auxiliar na descrição a arquitetura de cada cenário proposto na etapa anterior.
  - Descrição formal da arquitetura dos cenários: Com base no mapeamento realizado e de acordo com as regras da linguagem descreveremos a arquitetura dos cenários em um *editor de código π-ADL*.
- Na etapa de verificação foram realizados os seguintes passo:
  - Especificação do modelo: Através da descrição da arquitetura em π-ADL é gerado um código Go, que compilado gera um executável, onde esses executável é a entrada necessária.
  - Especificação das propriedades que iremos avaliar: Nesse passo as propriedades são descritas usando DynBLTL.
  - Realização da análise: Executamos a verificação dos modelos na ferramenta para obtermos os resultados.

Mais detalhes referentes a realização destas etapas são apresentadas no capitulo a seguir.

Capítulo 4

SOLUÇÃO PROPOSTA

Neste capítulo é apresentado o que foi realizado buscando elucidar o problema da detecção de erros em tempo de execução em SoS modelados em diagramas de coreografia BPMN. Descrevemos detalhadamente as etapas da nossa proposta de solução, assim com a realização das mesmas. Na Seção 4.1 apresentamos a visão geral da solução proposta, na Seção 4.2 apresentamos a caracterização dos erros em diagramas de coreografia e na Seção 4.3 é apresentada a realização das etapas em subseções, sendo a apresentação do cenário, apresentação do mapeamento e da etapa de descrição e verificação, buscando exemplificar a proposta.

4.1 APRESENTAÇÃO DA SOLUÇÃO PROPOSTA

Nosso objetivo foi propor um método para identificar erros que ocorrem em tempo de execução, como os deadlocks e livelocks, em Sistema de Sistemas modelados utilizando o diagrama de coreografia da notação BPMN, a fim de garantir a ausência desses erros. Um erro de deadlock, por exemplo, ocorre quando os processos estão aguardando recursos, causando travamento, impedindo que eles possam ser adequadamente fechados ou concluídos. Nossa abordagem consiste em oferecer um modelo formal, baseado na linguagem π-ADL, para BPMN que permita a verificação desse modelo.

Essa abordagem parte da modelagem visual dos processos de negócios, utilizando o diagrama de coreografia proposto em BPMN 2.0, até sua análise usando um verificador de modelos específico. Como BPMN carece de uma semântica formal, foi necessário estabelecer um mapeamento com a tradução dos elementos da notação BPMN para linguagem formal π-ADL. Este mapeamento consiste da descrição, com base na semântica da linguagem π-ADL, do comportamento de cada elemento do diagrama de coreografia individualmente, para auxiliar na tradução de sistemas modelados nessa notação. Fazemos uso de ferramentas que permitem a validação da especificação formal, e também a realização da verificação de modelos em um verificador específico para a linguagem π-ADL.

Para a realização das etapas descritas na Figura 3.2 (ver Seção 3.3) foi utilizado uma cadeia de ferramentas que oferece suporte para investigar arquiteturas descritas em π-ADL. Essa cadeia de ferramentas é composta de 2 ferramentas e ambas foram utilizadas no decorrer desse trabalho. A primeira é o editor de código π-ADL, um plug-in para o Eclipse que foi utilizado na etapa de descrição, que gera um código fonte na linguagem de programação Go baseado na descrição π-ADL feita, código esse que serve como base para entrada da segunda ferramenta. A segunda ferramenta é o PLASMA, utilizada na etapa de verificação, que é equipado com plug-ins que interagem com o executável gerado com a compilação do código Go obtido na etapa anterior e verifica as propriedades escritas em DynBLTL.

4.2 CARACTERIZAÇÃO DOS ERROS

Com base na caracterização apresentada em (KHERBOUCHE; AHMAD; BASSON, 2012) e (KHERBOUCHE; AHMAD; BASSON, 2013), definimos a caracterização dos deadlocks e livelocks que podem ocorrer nos diagramas de coreografia BPMN e que usaremos para ilustrar a proposta. Estes exemplos podem ser os seguintes:

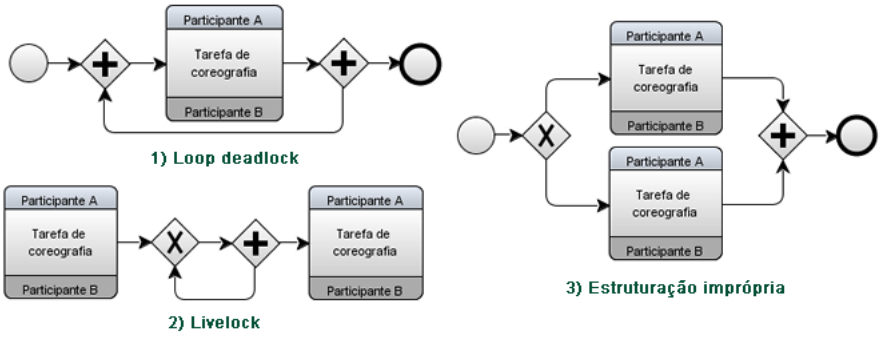


Figura 4.1 Caracterização dos erros em diagrama de coreografia BPMN.

- Deadlock ocorre das seguintes formas:
  - Loop deadlock ocorre quando há um caminho de execução da saída de uma junção *AND* para seus pontos de entrada. Se esse caminho não contiver partições *OR*, há ocorrência de deadlock, conforme a Figura 4.1 (1).
  - Estruturação incorreta ocorre quando junção *AND*, recebe entrada partições *OR*, conforme a Figura 4.1 (3).
- Livelock pode dar em uma execução infinita do processo. Nesse caso, parte do processo pode ser executado com sucesso, mas alguns podem ser bloqueados em um loop infinito de execução, conforme a Figura 4.1 (2).

4.3 EXEMPLIFICAÇÃO DA PROPOSTA

Como validação inicial da abordagem, definimos um cenário no contexto de SoS modelado usando o diagrama de coreografia, tendo situações de deadlock. Em seguida, na Seção 4.3.1 mostra o cenário usado nessa validação, a Seção 4.3.2 mostra a proposta de mapeamento e a Seção 4.3.3 apresenta um exemplo de uso do mapeamento para realizar a descrição do cenário em questão. Por fim a seção 4.3.4 mostra a realização da etapa de verificação do modelo.

4.3.1 Definição do cenário

Seguindo os passos definidos para a realização da etapa de descrição, começamos desenvolvendo o cenário que foi utilizado na experimentação para buscar validar a solução proposta. Partimos da criação de um cenário mais simples que representa um processo de serviço realizado em uma Smart Home, conforme mostra a Figura 4.2.

Na Figura 4.2 é apresentado um diagrama criado com base no trabalho (XU et al., 2012) que se trata de um cenário no contexto de Smart Home. O trabalho apresenta uma nova ideia de casa inteligente que amplia a ideia tradicional, onde sistema de serviço de casa inteligente (SHS) pode ser definido como um sistema inteligente voltado para a vida familiar e as instalações domésticas. A arquitetura do SHS constituída por três redes: IoT de casa inteligente, rede de serviço de casa inteligente e rede social da família, seu objetivo é agregar essas redes para fornecer serviços inteligentes.

SHS inclui os principais elementos do servidor front-end, centros de serviços domésticos, usuários familiares, provedores de serviços, instalações domésticas e nuvem doméstica inteligente. Os usuários familiares podem usar os serviços domésticos locais ou os serviços no centro de serviços remoto por meio das interfaces fornecidas pelo servidor front-end, que geralmente é empacotado como uma caixa, por isso chamamos de "Smart Magic Box (SMB)".

No contexto de Smart-home os domínios de negócios são variados, como: segurança, compras, saúde, entretenimento, entre outros e cada um deles também inclui uma variedade de funções de serviço. No trabalho é proposto um método de modelagem baseado em BPMN para estabelecer os modelos de processos de negócios para SHS, onde os processos de negócios são analisados e classificados e, em seguida, é apresentado um negócio de compras como exemplo, o qual nos baseamos para definir nosso diagrama de coreografia.

O negócio de compras está relacionado a quatro tipos de participantes: usuários familiares, SMC, centro de serviços de compras e provedores de serviços. E as interações são principalmente entre usuários familiares e centro de serviços, centro de serviços e provedores de serviços.

O processo ocorre da seguinte forma: quando o SMB detecta a falta de mercadorias, o sistema notifica aos usuários da família que em seguida podem, ou não, enviar uma requisição de compra ao centro de serviço. Ao decidir fazer o pedido, os usuários da família podem gerar pedido semelhante a pedidos anteriores, com referência aos registros do histórico, ou escolher mercadorias diretamente na lista de detalhes.

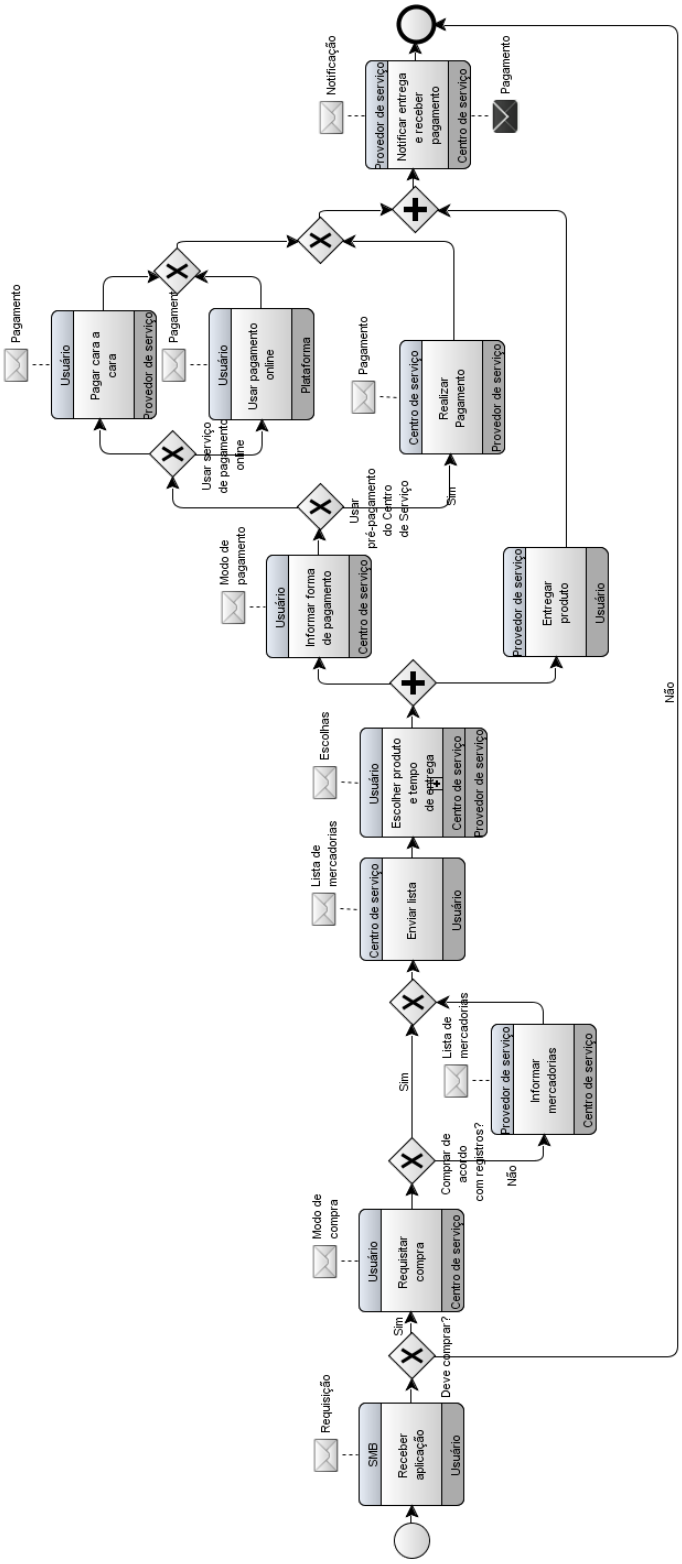


Figura 4.2 Diagrama de coreografia de serviço realizado em uma Smart Home.

Depois que todas as informações forem preenchidas, o pedido será enviado ao centro de serviços de compras. O centro de serviço de compras analisa o pedido e sendo aprovados serão enviados aos prestadores de serviços correspondentes, ou seja, aos fornecedores. De acordo com o pedido, os prestadores de serviços entregam as mercadorias aos usuários, recebem o pagamento, notifica ao centro de serviço a entrega e o processo é finalizado.

Tendo definido o cenário a ser utilizado, partimos para a passo seguinte que foi o mapeamento dos elementos, etapa essa é apresentada na subseção que segue.

4.3.2 Mapeamentos dos elementos do diagrama de coreografia BPMN 2.0

Dando seguimento aos passos propostos, descrevemos com base na semântica da linguagem π-ADL o comportamento de cada elemento do diagrama de coreografia individualmente para, na sequência poder auxiliar, também de acordo como propõe a linguagem, a compor a especificação do cenário. Detalhamos alguns desses elementos a seguir:

- Evento de início: o ponto onde o processo inicia, seu comportamento é o envio do start do processo para seus elementos vizinhos, através da saída do evento.
- Fluxo de sequência: implica uma dependência entre as atividades conectadas, seu comportamento é receber dados através da entrada e os enviar através da saída.
- Tarefa de coreografia: onde descreve processos de troca de mensagens. Ao receber uma entrada, inicializa interações entre os participantes que ao fim dessa interação envia dados através da saída.
- Gateways: criam caminhos alternativos ou paralelos, seu comportamento depende do seu tipo.
  - Exclusivo: para separar o fluxo, ao receber entrada de dados, o processo deverá verificar a condição indicada, e apenas uma das saídas do gateway dará seguimento. E para unificar, a primeira entrada que chegar dará continuidade no fluxo do processo, através da saída.
  - Paralelo: para dividir o fluxo, o receber entrada de dados, todos os caminhos que saem deste gateway são executados. Na convergência, ele garantirá que receba todas as entradas para dar continuidade ao fluxo de saída.
- Evento de fim: o ponto onde o processo termina, seu comportamento é receber dados de um elemento vizinho via entrada do evento, na sequência encerrar o processo.

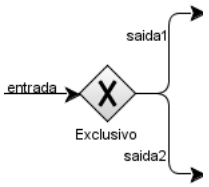
Realizamos o mapeamento com a descrição dos elementos apresentados acima, que são alguns elementos principais de modelagem BPMN, e que estão presentes no cenário apresentado na seção anterior. Nosso mapeamento pode ser observado na Tabela 4.1, ele é apenas com relação parâmetro *behavior* (se refere ao comportamento do elemento), pois com base nessa descrição podemos extrair os dados para descrever os demais parâmetros que compõe a descrição dos componentes e conectores.

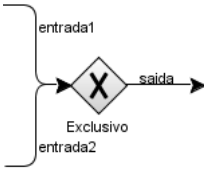
Elemento BPMN	Tradução do comportamento em π-ADL

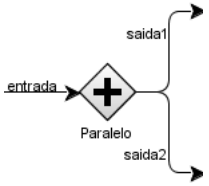
	`behavior is{`
	`  via saida send`
	`  behavior()`
	`}`

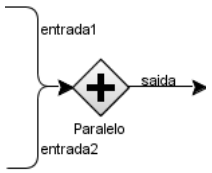
	`behavior is {`
	`  via entrada receive`
	`  behavior()`
	`}`

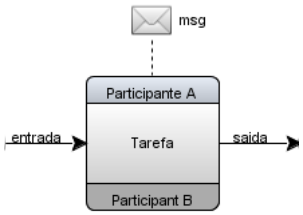
	`behavior is {`
	`  via deComp receive`
	`  via paraComp send`
	`  behavior()`
	`}`

	`behavior is {`
	`  via entrada receive`
	`  if c1 then{`
	`  via saida1 send`
	`  behavior()`
	`  }`
	`  else{`
	`  via saida2 send`
	`  behavior()`
	`  }`
	`}`

	`behavior is{`
	`  choose {`
	`  via entrada receive`
	`  via saida send`
	`  behavior()`
	`  or`
	`  via entrada2 receive`
	`  via saida send`
	`  behavior()`
	`  }`
	`}`

	`behavior is{`
	`  via entrada receive`
	`  compose {`
	`  via saida1 send`
	`  and`
	`  via saida2 send`
	`  }`
	`}`

	`behavior is{`
	`  compose {`
	`  via entrada1 receive`
	`  and`
	`  via entrada2 receive`
	`  }`
	`  via saida send`
	`  behavior ()`
	`}`

	`behavior is {`
	`  via entrada receive`
	`  unobservable`
	`  via participanteA receive`
	`  via participanteB send`
	`  via saida send`
	`  behavior()`
	`}`

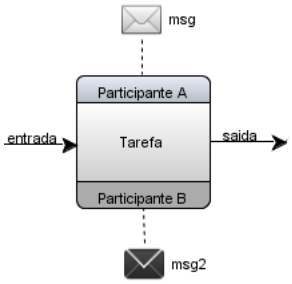
	`behavior is {`
	`  via entrada receive`
	`  via participanteA receive`
	`  via participanteB send`
	`  via participanteB2 receive`
	`  via participanteA2 send`
	`  via saida send`
	`  behavior()`
	`}`

Tabela 4.1: Mapeamento dos elementos BPMN para π-ADL

4.3.3 Descrições da arquitetura dos cenários

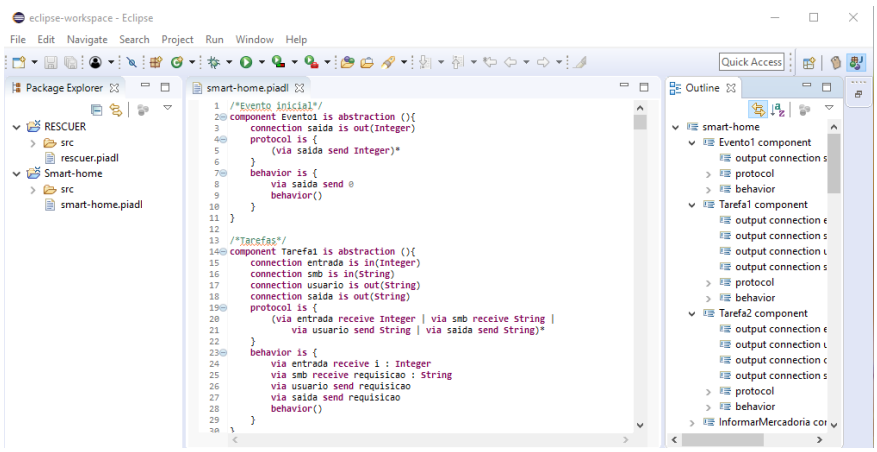


Figura 4.3 Tela do Editor textual π-ADL baseado no Eclipse.

Para a realização da descrição do processo realizado na Smart Home apresentado na Figura 4.2 utilizamos como base o mapeamento que foi apresentado na Tabela 4.1 na Seção anterior, resultando assim, a especificação formal do processo. Essa especificação do cenário foi realizada utilizando uma ferramenta de suporte para π-ADL apresentado no trabalho (CAVALCANTE, 2016).

Um Editor de textual, que é um plug-in para o Eclipse, desenvolvido, justamente para auxiliar na descrição de arquitetura utilizando a linguagem π-ADL, a Figura 4.3 apresenta a interface do editor textual. Nele é permitido fazer validação sintática e semântica enquanto realizamos a descrição, mostrando erros e permitindo assim a detecção e correção de e problemas potenciais, além de gerar automaticamente o código de implementação em Go.

A seguir apresentaremos as descrições formais de alguns dos elementos presentes no cenário da Figura 4.2.

```
1  component Inicio is abstraction (){
2      connection saida is out(Integer)
3      protocol is {
4          (via saida send Integer)*
5      }
6      behavior is {
7          via saida send 0
8          behavior()
9      }
10 }
```

Figura 4.4 Descrição em π-ADL do componente *Inicio*.

Componente Inicio: a Figura 4.4 mostra a especificação do componente *Inicio* que é um evento de início, onde ocorre o start do processo com a notificação ao usuário feita pelo SMB, representado através da variável *saida*, que ao ser enviada inicializa o processo.

```
1  connector Fluxo is abstraction (){
2      connection deInicio is in(Integer)
3      connection paraReceberAplicacao is out(Integer)
4      protocol is {
5          (via deInicio receive Integer | via paraReceberAplicacao send Integer)*
6      }
7      behavior is {
8          via deInicio receive x : Integer
9          via paraReceberAplicacao send x
10         behavior()
11     }
12 }
```

Figura 4.5 Descrição em π-ADL do conector *Fluxo1*.

Conector Fluxo 1: a Figura 4.5 representa o conector *Fluxo* que nesse cenário serve como conexão entre o componente *Inicio* e o componente *ReceberAplicacao*, ou seja, o envio dos valores entre os componentes em questão.

```
component ReceberAplicacao is abstraction (){
    connection entradaT1 is in(Integer)
    connection smb is in(String)
    connection usuario is out(String)
    connection saidaT1 is out(String)
    protocol is {
        (via entradaT1 receive Integer | via smb receive String |
            via usuario send String | via saidaT1 send String)*
    }
    behavior is {
        via entradaT1 receive i : Integer
        via smb receive requisicao : String
        via usuario send requisicao
        via saidaT1 send requisicao
        behavior()
    }
}
```

Figura 4.6 Descrição em π-ADL do componente *ReceberAplicacao*.

Componente ReceberAplicacao: a Figura 4.6 apresenta a descrição do componente *ReceberAplicacao* que representa a atividade de coreografia do processo, onde mostra a troca de informação entre os participantes, nesse caso, o usuário recebendo a notificação do *SMB*, que é declarado como *requisicao* na descrição abaixo.

Após a descrição de todos os elementos do diagrama de coreografia, devemos descrever o comportamento de cada um dos participantes e suas conexões com cada tarefa de coreografia que participam. A seguir exemplificaremos a descrição formal de um participante e sua conexão do cenário em questão, isso se aplica a todos os participantes e com as tarefas as quais participam.

Participante SMB: a Figura 4.7 apresenta o participante *SMB* que faz parte da primeira tarefa de coreografia presente no diagrama, chamada *ReceberAplicacao*. O participante *SMB* realiza o envio de uma mensagem para outro participante através dessa tarefa de coreografia.

```
component Smb is abstraction(){
    connection smb is out(String)
    protocol is {
        (via smb send String)*
    }
    behavior is {
        via smb send "requisicao"
        behavior()
    }
}
```

Figura 4.7 Descrição em π-ADL do participante *SMB*.

Conector Conexão: a Figura 4.8 mostra o conector *conexao* que conecta o participante *SMB* e a tarefa de coreografia *ReceberAplicacao*.

```
connector conexao is abstraction (){
    connection deSmb is in(String)
    connection paraReceberAplicacao is out(String)
    protocol is {
        (via deSmb receive String | via paraReceberAplicacao send String)*
    }
    behavior is {
        via deSmb receive x : String
        via paraReceberAplicacao send x
        behavior()
    }
}
```

Figura 4.8 Descrição em π-ADL do conector *Conexao*.

Ao fim da descrição de todos os elementos e todos os participantes e suas conexões é realizada a descrição da arquitetura do cenário, onde todos eles são apresentados juntamente com suas interações.

Arquitetura Smart Home: A Figura 4.9 apresenta a descrição completa da arquitetura da Figura 4.2, que é composta por dois eventos, um inicial e um final, onze atividades de coreografia, nove gateways e vinte e seis fluxos fazendo a conexão entre todos os elementos. Essa descrição representa o fluxo de dados de um elemento para o outro, onde transmite dados desde o evento início até o evento final.

```
architecture SmartHome is abstraction () {
    behavior is {
        compose{
            i is Inicio()
            and t1 is ReceberAplicacao()
            and t2 is RequisitarCompra()
            and t3 is InformarMercadoria()
            and t4 is EnviarLista()
            and t5 is EscolherProduto()
            and t6 is EntregarProduto()
            and t7 is InformarPagamento()
            and t8 is PagarDinheiro()
            and t9 is PagarOnline()
            and t10 is RealizarPagamento()
            and t11 is NotificarEntrega()
            and g1 is Gateway1()
            and g2 is Gateway2()
            and g3 is Gateway3()
            and g4 is Gateway4()
            and g5 is Gateway5()
            and g6 is Gateway6()
            and g7 is Gateway7()
            and g8 is Gateway8()
            and g9 is Gateway9()
            and fl1 is Fluxo()
            and fl2 is Fluxo2()
            and fl3 is Fluxo3()
            and fl4 is Fluxo4()
            and fl5 is Fluxo5()
            and fl6 is Fluxo6()
            and fl7 is Fluxo7()
            and fl8 is Fluxo8()
            and fl9 is Fluxo9()
            and fl10 is Fluxo10()
            and fl11 is Fluxo11()
            and fl12 is Fluxo12()
            and fl13 is Fluxo13()
            and fl14 is Fluxo14()
            and fl15 is Fluxo15()
            and fl16 is Fluxo16()
            and fl17 is Fluxo17()
            and fl18 is Fluxo18()
            and fl19 is Fluxo19()
            and fl20 is Fluxo20()
            and fl21 is Fluxo21()
            and fl22 is Fluxo22()
            and fl23 is Fluxo23()
            and fl24 is Fluxo24()
            and fl25 is Fluxo25()
            and fl26 is Fluxo26()
            and fn is Fim ()
            and p1 is Smb()
            and p2 is Usuario()
            and p3 is CentroServico()
            and p4 is ProvedorServico()
            and p5 is Plataforma()
            and c1 is conexao()
            and c2 is conexao2()
            and c3 is conexao3()
            and c4 is conexao4()
            and c5 is conexao5()
            and c6 is conexao6()
            and c7 is conexao7()
            and c8 is conexao8()
```

```
            and c9 is conexao9()
            and c10 is conexao10()
            and c11 is conexao11()
            and c12 is conexao12()
            and c13 is conexao13()
            and c14 is conexao14()
            and c15 is conexao15()
            and c16 is conexao16()
            and c17 is conexao17()
            and c18 is conexao18()
            and c19 is conexao19()
            and c20 is conexao20()
            and c21 is conexao21()
            and c22 is conexao22()
            and c23 is conexao23()
            and c24 is conexao24()
            } where {
            i::saida unifies fl1::deInicio
            fl1::paraReceberAplicacao unifies t1::entradaT1
            t1::saidaT1 unifies fl2::deReceberAplicacao
            fl2::paraGateway1 unifies g1::entradaG1
            g1::saida1G1 unifies fl3::deGateway1
            fl3::paraFim unifies fn::entrada
            g1::saida2G1 unifies fl4::deGateway1
            fl4::paraRequisitarCompra unifies t2::entradaT2
            t2::saidaT2 unifies fl5::deRequisitarCompra
            fl5::paraGateway2 unifies g2::entradaG2
            g2::saida1G2 unifies fl6::deGateway2
            fl6::paraInformarMercadoria unifies t3::entradaT3
            g2::saida2G2 unifies fl7::deGateway2
            fl7::paraGateway3 unifies g3::entradaG3
            t3::saidaT3 unifies fl8::deInformarMercadoria
            fl8::paraGateway3 unifies g3::entrada2G3
            g3::saidaG3 unifies fl9::deGateway3
            fl9::paraEnviarLista unifies t4::entradaT4
            t4::saidaT4 unifies fl10::deEnviarLista
            fl10::paraEscolherProduto unifies t5::entradaT5
            t5::saidaT5 unifies fl11::deEscolherProduto
            fl11::paraGateway4 unifies g4::entradaG4
            g4::saida1G4 unifies fl12::deGateway4
            fl12::paraEntregarProduto unifies t6::entradaT6
            t6::saidaT6 unifies fl14::deEntregarProduto
            fl14::paraGateway9 unifies g9::entrada2G9
            g4::saida2G4 unifies fl13::deGateway4
            fl13::paraInformarPagamento unifies t7::entradaT7
            t7::saidaT7 unifies fl15::deInformarPagamento
            fl15::paraGateway5 unifies g5::entradaG5
            g5::saida1G5 unifies fl16::deGateway5
            fl16::paraRealizarPagamento unifies t10::entradaT10
            t10::saidaT10 unifies fl18::deRealizarPagamento
            fl18::paraGateway8 unifies g8::entrada2G8
            g5::saida2G5 unifies fl17::deGateway5
            fl17::paraGateway6 unifies g6::entradaG6
            g6::saida1G6 unifies fl19::deGateway6
            fl19::paraPagarDinheiro unifies t8::entradaT8
            t8::saidaT8 unifies fl21::dePagarDinheiro
            fl21::paraGateway7 unifies g7::entradaG7
            g6::saida2G6 unifies fl20::deGateway6
            fl20::paraPagarOnline unifies t9::entradaT9
            t9::saidaT9 unifies fl22::dePagarOnline
            fl22::paraGateway7 unifies g7::entrada2G7
            g7::saidaG7 unifies fl23::deGateway7
            fl23::paraGateway8 unifies g8::entradaG8
            g8::saidaG8 unifies fl24::deGateway8
            fl24::paraGateway9 unifies g9::entradaG9
            g9::saidaG9 unifies fl25::deGateway9
            fl25::paraNotificarEntrega unifies t11::entradaT11
            t11::saidaT11 unifies fl26::deNotificarEntrega
            p1::smb unifies c1::deSmb
            c1::paraReceberAplicacao unifies t1::smb
            t1::usuario unifies c2::deReceberAplicacao
            c2::paraUsuario unifies p2::usuario
            p2::usuario2 unifies c3::deUsuario
            c3::paraRequisitarCompra unifies t2::usuario2
            t2::centroservico unifies c4::deRequisitarCompra
            c4::paraCentroServico unifies p3::centroservico
            p4::provedorservico unifies c5::deProvedorServico
            c5::paraInformarMercadoria unifies t3::provedorservico
            t3::centroservico2 unifies c6::deInformarMercadoria
            c6::paraCentroServico unifies p3::centroservico2
            p3::centroservico3 unifies c7::deCentroServico
            c7::paraEnviarLista unifies t4::centroservico3
            t4::usuario3 unifies c8::deEnviarLista
            c8::paraUsuario unifies p2::usuario3
            p2::usuario4 unifies c9::deUsuario
            c9::paraEscolherProduto unifies t5::usuario4
            t5::centroservico4 unifies c10::deEscolherProduto
            c10::paraCentroServico unifies p3::centroservico4
            p3::centroservico5 unifies c11::deCentroServico
            c11::paraEscolherProduto unifies t5::centroservico5
            t5::provedorservico2 unifies c12::deEscolherProduto
            c12::paraProvedorServico unifies p4::provedorservico2
            p4::provedorservico3 unifies c13::deProvedorServico
            c13::paraEntregarProduto unifies t6::provedorservico3
            t6::usuario5 unifies c14::deEntregarProduto
            c14::paraUsuario unifies p2::usuario5
            p2::usuario6 unifies c15::deUsuario
            c15::paraInformarPagamento unifies t7::usuario6
            t7::centroservico6 unifies c16::deInformarPagamento
            c16::paraCentroServico unifies p3::centroservico6
            p2::usuario7 unifies c17::deUsuario
            c17::paraPagarDinheiro unifies t8::usuario7
            t8::provedorservico5 unifies c18::dePagarDinheiro
            c18::paraProvedorServico unifies p4::provedorservico5
            p2::usuario8 unifies c19::deUsuario
            c19::paraPagarOnline unifies t9::usuario8
            t9::plataforma unifies c20::dePagarOnline
            c20::paraPlataforma unifies p5::plataforma
            p3::centroservico7 unifies c21::deCentroServico
            c21::paraRealizarPagamento unifies t10::centroservico7
            t10::provedorservico4 unifies c22::deRealizarPagamento
            c22::paraProvedorServico unifies p4::provedorservico4
            p4::provedorservico6 unifies c23::deProvedorServico
            c23::paraNotificarEntrega unifies t11::provedorservico6
            t11::centroservico8 unifies c24::deNotificarEntrega
            c24::paraCentroServico unifies p3::centroservico8
            fl26::paraFim unifies fn::entrada
        }
    }
}
behavior is {
    become(SmartHome())
}
            }
```

Figura 4.9 Descrição em π-ADL da arquitetura parte inicial.

4.3.4 Realização da etapa de verificação

Nessa etapa buscamos realizar a verificação, com o auxílio de um modelo de verificação, para aferir se a formalização na linguagem π-ADL do cenário em BPMN, ajuda na detecção de possíveis erros que podem ocorrem na execução do sistema modelado. Descrevemos a seguir como realizar a verificação estatística das descrições π-ADL.

Como já foi citado anteriormente, nosso trabalho usamos as técnicas de Verificação Estatística de Modelos (SMC), que é uma técnica probabilística baseada em simulação

destinada a verificar, se uma determinada propriedade é satisfeita durante a execução de um sistema, ou seja, consideram os valores verdadeiros.

Para a realização da verificação utilizamos a ferramenta PLASMA, devido ao fato dela dar suporte para investigar a probabilidade de arquiteturas descritas em π-ADL satisfazer determinadas propriedades especificada no DynBLTL.

Para obter as entradas que alimentaram a ferramenta PLASMA, realizamos: (i) Especificação do modelo: Ao salvar o arquivo.piadl da descrição da arquitetura apresentada parcialmente na seção anterior, é acionada a geração do código Go a partir desse arquivo, que compilado gera um executável, que é uma das entrada necessária. Essa compilação geralmente é realizada de forma manual em um terminal Linux e (ii) Especificação das propriedades que iremos avaliar: Nesse passo as propriedades são descritas usando DynBLTL. A formúla referente a propriedade é inserida diretamente no PLASMA em um campo definido especificamente para tal.

4.3.4.1 Definição das propriedades Conforme citamos anteriormente precisamos definir as propriedades que serviram para verificar a ausência de deadlock nos modelos apresentados, e para isso utilizamos a lógica DynBLTL. É uma lógica para expressar propriedades temporais lineares em relação a traços de sistemas dinâmicos, pois o sistema de software dinâmico tem como característica a impossibilidade de prever o conjunto exato de elementos arquitetônicos implantados em um determinado ponto de execução. Foi projetada justamente pelo fato de outras lógicas, como por exemplo a BLTL, são incapazes de lidar com a característica em questão. Sua sintaxe é apresentada em (QUILBEUF et al., 2016).

Para realização das verificações precisamos definir as propriedades para serem avaliadas. Uma das formas que deadlocks podem ocorrer é devido ao mal posicionamento de gateways, é possível expressar propriedades para assegurar seu correto funcionamento, o que garante que ele está no local correto. Assumindo que um gateway exclusivo de junção é correto se recebe dados por uma das duas portas de entradas, tal declaração pode ser definida como:

```
always during X time units {
 forall g:allOfType(Gateway) {
  isTrue (g.entrada1 > 0 or g.entrada2 > 0) implies
  (eventually before Y time units g.saida > 0)
 }
}
```

Essa propriedade expressa que o recebimento de dados por uma das entradas do gateway exclusivo implica que os dados precisam ser enviados através da porta de saída em um espaço de tempo definido através do limite representado por Y. O limite representado por X na formula define as unidades de tempo finito que a propriedade pode ser decidida.

```
always during X time units {
 forall g:allOfType(Gateway) {
  isTrue (g.entrada1 > 0 and g.entrada2 > 0) implies
  (eventually before Y time units g.saida > 0)
 }
```

}

Onde, a propriedade expressa que o recebimento de dados por ambas as entradas do gateway paralelo implica que os dados precisam ser enviados através da porta de saída igualmente definido acima. No próximo capitulo apresentamos mais algumas propriedades verificadas.

4.3.4.2 Executando a verificação Para os resultados, executamos a verificação das propriedades do modelo na ferramenta para obtermos os resultados. Inicialmente devemos criar um novo projeto na plataforma PLASMA, onde entraremos com o caminho do executável e com as propriedades que serão verificadas.

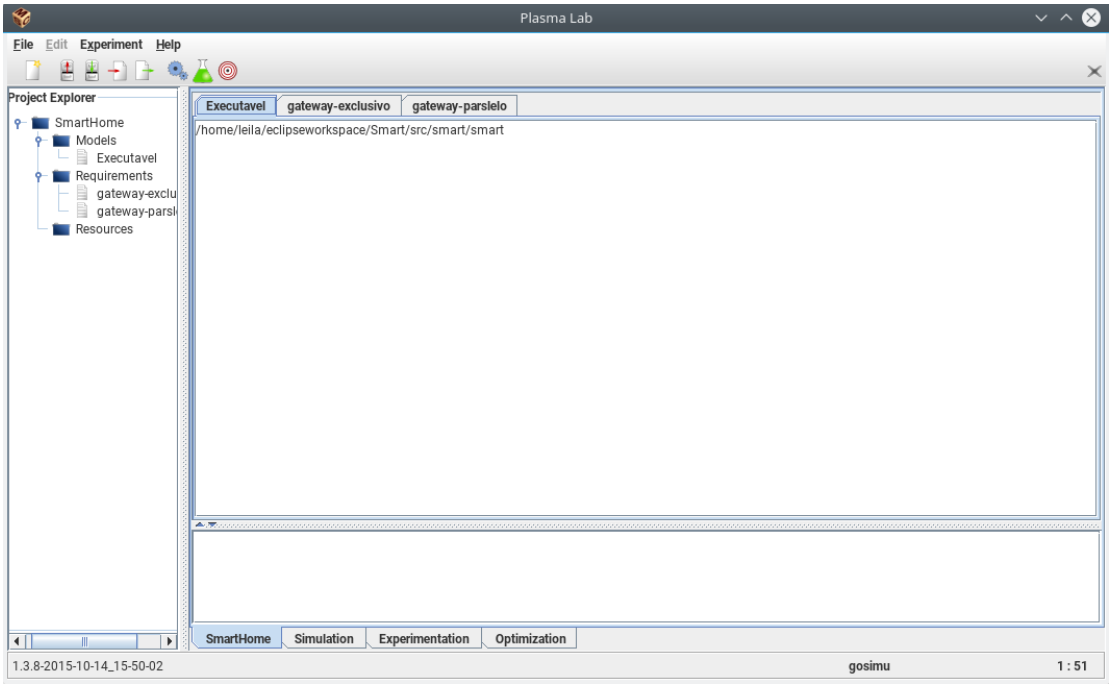


Figura 4.10 Captura da tela da plataforma PLASMA.

A Figura 4.10 apresenta a captura da tela da plataforma PLASMA, onde se observa o projeto criado e as entradas necessárias inseridas.

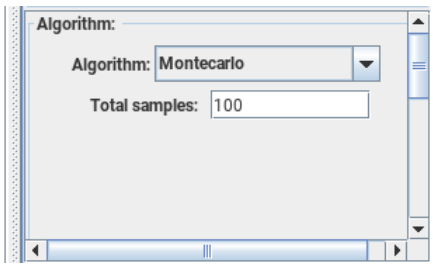


Figura 4.11 Definição do número de simulações com algoritmo Monte Carlo.

Para essa verificação usamos o algoritmo Monte Carlo, pois seus métodos permitem estimar a probabilidade de satisfazer um requisito usando limites de confiança. São usados para calcular o valor médio de uma propriedade ocorrer, retornando um valor resultante de alguns cálculos ao longo do rastreamento. Para executa-lo, precisamos apenas definir o número de simulações a serem executadas, como observamos na Figura 4.11. Este método atende bem a nossa necessidade de saber se uma propriedade ocorre ou não em nosso modelo.

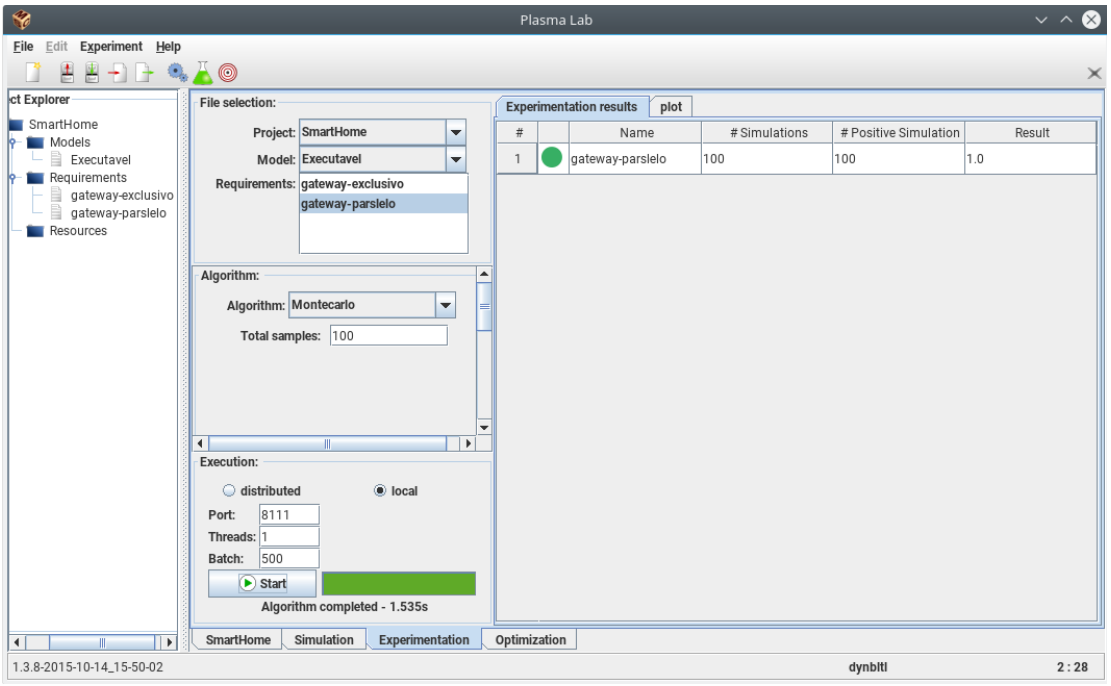


Figura 4.12 Captura da tela de experimentação da plataforma PLASMA.

Após realizarmos a verificação da propriedade podemos observar facilmente os resultados da experimentação, constatar a existência ou não do erro em um período determinado, como apresenta a Figura 4.12.

No próximo capitulo apresentamos a validação de todos os passos exemplificados neste capítulo, realizando todas as etapas seguindo até a experimentação com a análise completa dos resultados.

Capítulo 5

VALIDAÇÃO DA PROPOSTA

Neste capítulo tem aplicação da solução proposta onde executamos todo o processo proposto e apresentado na seção de exemplificação, mas com a intenção de validar a nosso trabalho. Neste capítulo, apresentaremos um cenário mais complexo, conforme é apresentado na Seção 5.1, na Seção 5.2 é apresentada a sua descrição formal na Seção 5.3 trazemos as propriedades que serão verificadas. Na Seção 5.4 a verificação formal e, por fim, apresentamos na Seção 5.5 a análise dos resultados.

5.1 EXEMPLIFICAÇÃO

5.1.1 Apresentação do cenário

O nosso cenário foi criado baseado no projeto RESCUER (VILLELA et al., 2018), que é um projeto que desenvolveu um sistema inteligente e interoperável de apoio à decisão para apoiar o gerenciamento de emergências e crises com base em informações de crowdsourcing. Foco do projeto estava em incidentes ocorridos em áreas industriais ou em eventos de grande escala, mas a plataforma resultante pode ser estendida para lidar com situações críticas em outros contextos. A Figura 5.1 ilustra a visão do RESCUER.

Ao observar a Figura 5.1 podemos perceber que os participantes, os atores na ocorrência de uma situação crítica são: (i) pessoas no local do incidente que causaram a situação crítica; (ii) unidades organizacionais encarregadas da segurança na área em que o incidente ocorreu, por ex. polícia, bombeiros e forças de resgate; (iii) membros das forças operacionais ou voluntários que são enviados ao local do incidente para controlar a emergência e restaurar as condições de segurança; (iv) grupo de pessoas designadas para avaliar riscos e tomar decisões em uma emergência e/ou crise em uma área industrial ou em um evento de grande escala;(v) pessoas na área afetada do incidente que provavelmente sofrerão com seu impacto e (vi) Imprensa, autoridades públicas e público em geral.

Para a criação do nosso cenário, assumimos um processo de incidentes ocorridos em áreas industriais, onde definimos participantes:

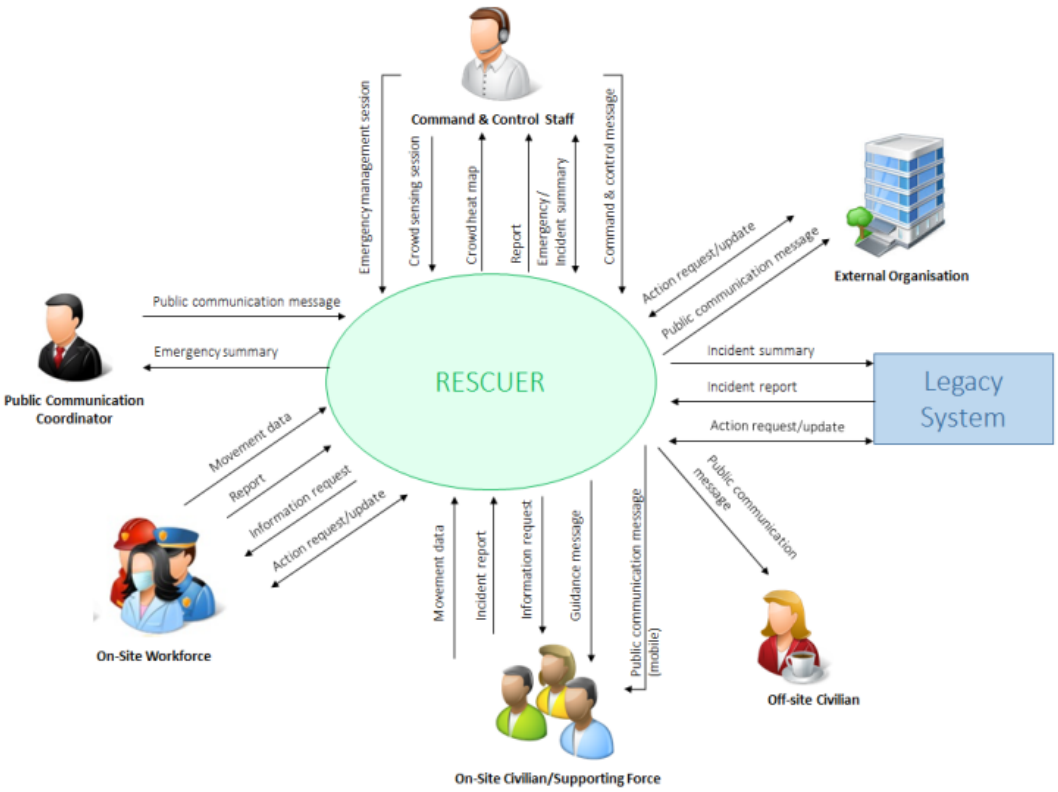


Figura 5.1 Visão macro do RESCUE (imagem apresentada em (VILLELA et al., 2018))

- Comando em controle;
- Civil interno;
- Civil externo;
- Coordenador de comunicação;
- Forças de apoio;
- Organizações externas.

Tudo começa com a equipe de Comando e Controle configurando a Sessão de Gerenciamento de Emergência, que define a área e o prazo para o gerenciamento de emergências. Várias sessões de detecção de multidões podem ser definidas dentro de uma sessão de gerenciamento de emergência para especificar quando e onde os dados de movimento devem ser coletados das pessoas no local (forças de trabalho / forças de apoio / civis). Isso é particularmente relevante no caso de eventos em grande escala que se espalham por grandes áreas e duram mais tempo.

Uma sessão de detecção de multidões resulta na criação de um mapa de calor da multidão, que usa cores para descrever a densidade de pessoas na área. Se nenhum

incidente for relatado, a Equipe de Comando e Controle simplesmente observa este Mapa de Calor da Multidão, que é atualizado periodicamente para mostrar a situação atual.

Com base na descrição apresentada acima, de como ocorre o processo no RESCUE, elaboramos um diagrama de orquestração como é apresentado na Figura 5.2, onde é apresentado de forma clara as interações que são realizadas estre os participantes. Após a conclusão deste diagrama partimos para a criação do nosso diagrama de coreografia, mostrado na Figura 5.3.

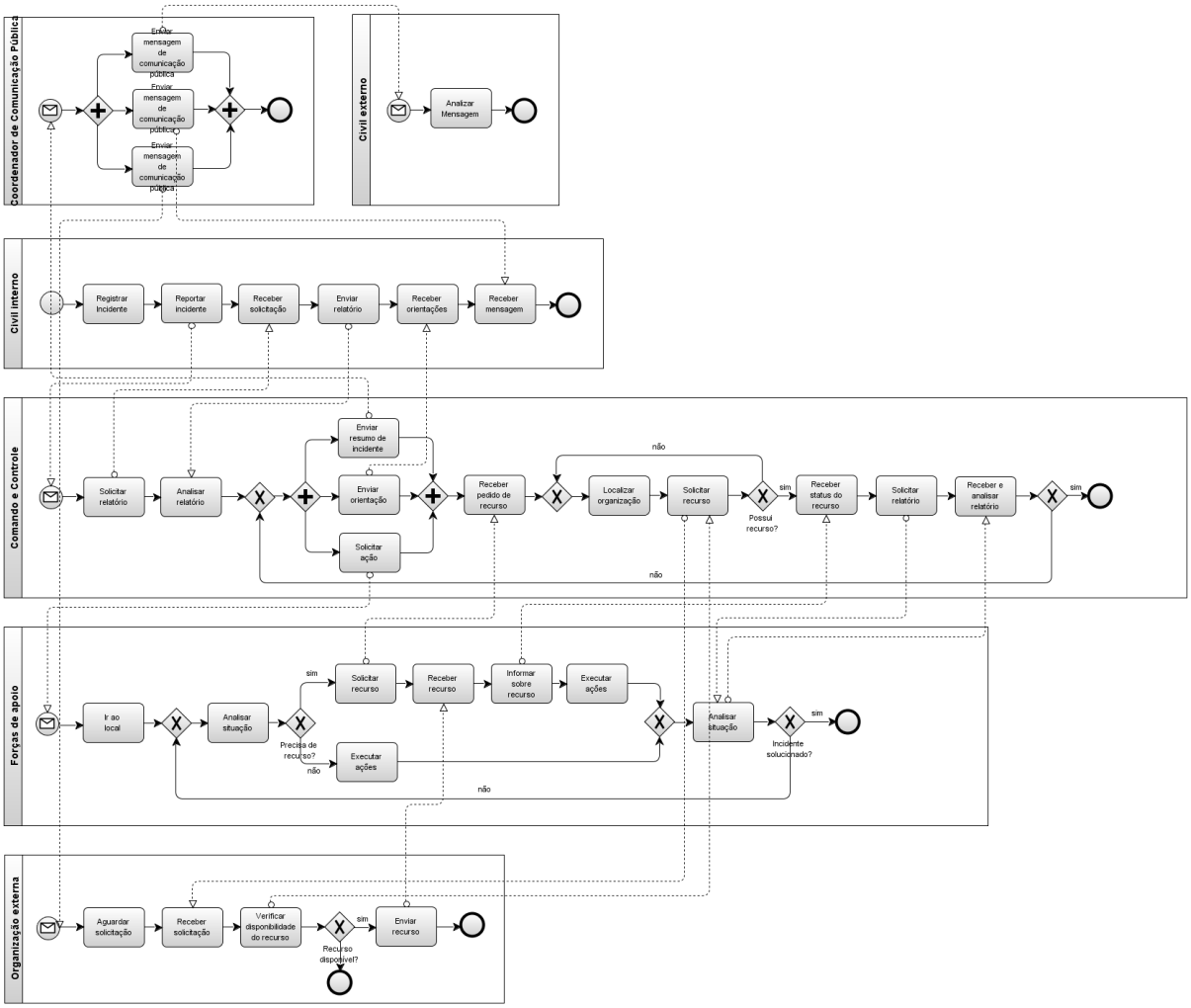


Figura 5.2 Diagrama de colaboração de incidente usando RESCUE.

5.1.2 Descrição da arquitetura do modelo

Conforme foi explicado e exemplificado na Subseção 4.2.3 do capítulo anterior, a descrição da arquitetura é apresentar como escrever o diagrama em termos de π-ADL. A seguir irei apresentar alguns dos elementos do diagrama anterior.

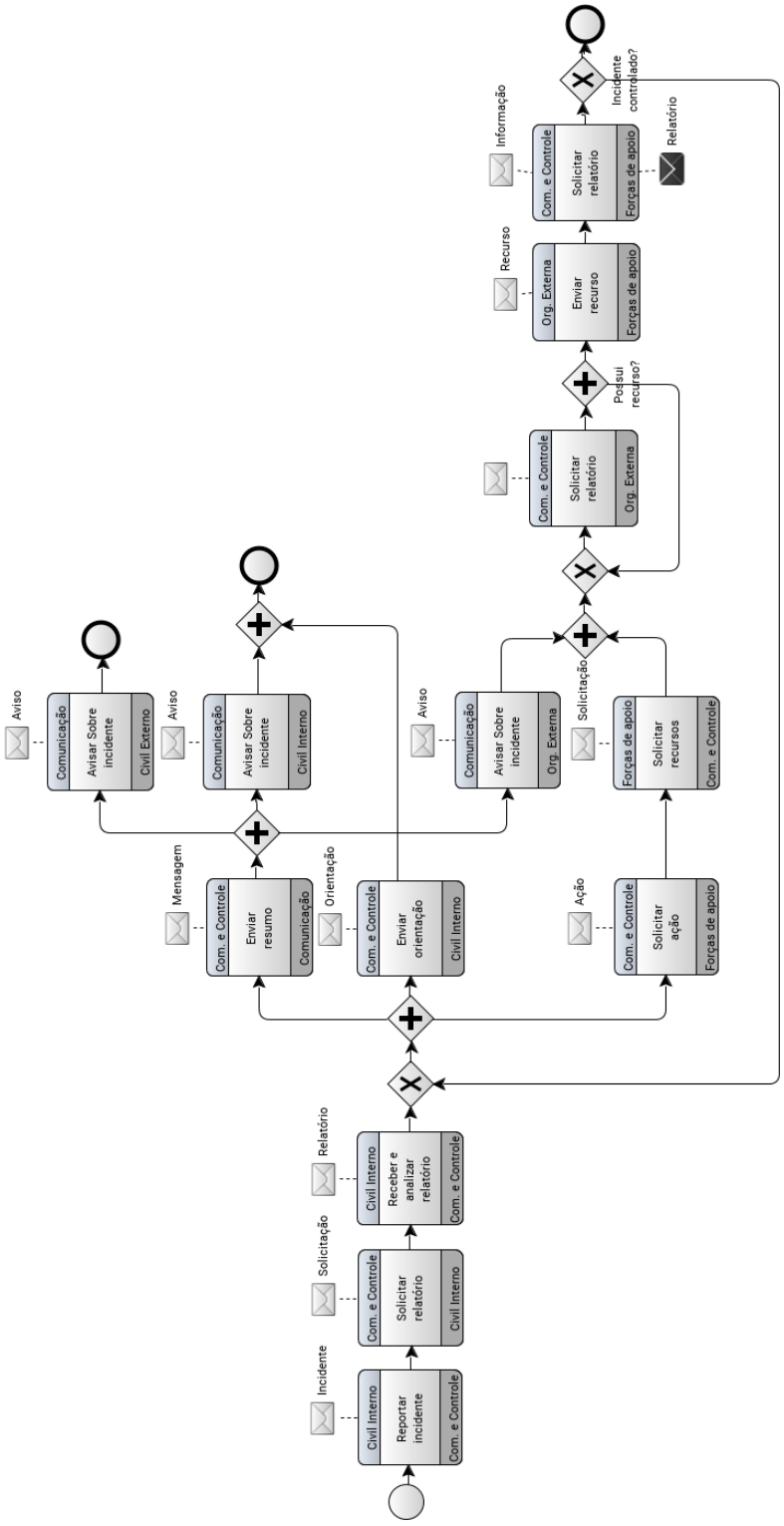


Figura 5.3 Diagrama de coreografia derivado do diagrama de colaboração.

Devido a extensão do código gerado na descrição do cenário apresentado na Figura 5.3, selecionei elementos mais variados apresentados no modelo em questão, buscando apresentar a grande variedade das descrições possíveis.

Inicialmente apresentarei o elemento de início, um fluxo de sequência e uma tarefa de coreografia, conforme foi feito no capítulo anterior. Seguindo da apresentação dos elementos antes não apresentados, como os gateways e o evento final.

Componente Inicio: a Figura 5.4 mostra a especificação do componente Inicio que é um evento de início, onde ocorre o start do processo com a notificação de ocorrência de incidente, feita pelo Civil interno para o Comando e controle, representado através da variável saida, que ao ser enviada inicializa o processo.

```
component Inicio is abstraction (){
    connection saida is out(Integer)
    protocol is {
        (via saida send Integer)*
    }
    behavior is {
        via saida send 0
        behavior()
    }
}
```

Figura 5.4 Descrição em π-ADL do componente Inicio.

Componente ReportarIncidente: a Figura 5.5 apresenta a descrição do componente ReportarIncidente que representa a tarefa de coreografia do processo, onde mostra a troca de informação entre os participantes, nesse caso, o comandocontrole recebendo a notificação do civilinterno, que é declarado como incidente na descrição abaixo. Todas as tarefas de coreografia seguem uma descrição semelhante, as considerando os participantes presentes nas tarefas e suas trocas de mensagens.

```
component ReportarIncidente is abstraction (){
    connection entradaT is in(Integer)
    connection civilinterno is in(String)
    connection comandocontrole is out(String)
    connection saidaT is out(String)
    protocol is {
        (via entradaT receive Integer | via civilinterno receive String |
            via comandocontrole send String | via saidaT send String)*
    }
    behavior is {
        via entradaT receive i : Integer
        via civilinterno receive incidente : String
        via comandocontrole send incidente
        via saidaT send incidente
        behavior()
    }
}
```

Figura 5.5 Descrição em π-ADL do componente ReportarIncidente.

Conector Fluxo: a Figura 5.6 representa o conector Fluxo que nesse cenário serve como conexão entre o componente Inicio e o componente ReportarIncidente, ou seja, o envio dos valores entre os componentes em questão. Todos os fluxos presentes no cenário possuem a mesma construção, considerando os elementos que realizam a conexão.

```
connector Fluxo is abstraction (){
    connection deInicio is in(Integer)
    connection paraReportarIncidente is out(Integer)
    protocol is {
        (via deInicio receive Integer | via paraReportarIncidente send Integer)*
    }
    behavior is {
        via deInicio receive x : Integer
        via paraReportarIncidente send x
        behavior()
    }
}
```

Figura 5.6 Descrição em π-ADL do conector Fluxo.

Componente Gateway1: a Figura 5.7 representa a descrição do componente Gateway1 que representa um gateway exclusivo de junção, onde há das duas entradas alternativas, representadas por entrada1G1 e entrada2G1 e apenas uma saída representada por saídaG1.

```
component Gateway1 is abstraction (){
    connection entrada1G1 is in(String)
    connection entrada2G1 is in(String)
    connection saidaG1 is out(String)
    protocol is {
        (via entrada1G1 receive String | via entrada2G1 receive String |
            via saidaG1 send String)*

    }
    behavior is {
        alerta is function(x: String) : String{
            return ("")
            }
        choose {
            via entrada1G1 receive relatorio : String
            via saidaG1 send alerta(relatorio)
            behavior()
            or
            via entrada2G1 receive relatorio2 : String
            via saidaG1 send alerta(relatorio2)
            behavior()
        }
    }
}
```

Figura 5.7 Descrição em π-ADL do componente Gateway1.

Componente Gateway2: a Figura 5.8 representa a descrição do componente Ga-

teway2 que representa um gateway paralelo, onde há uma entradaG2, representada por entrada e três saídas executas aos mesmo tempo, representada por saída1G2, saida2G2 e saida3G2.

```
component Gateway2 is abstraction (){
    connection entradaG2 is in(String)
    connection saida1G2 is out(String)
    connection saida2G2 is out(String)
    connection saida3G2 is out(String)
    protocol is {
        (via entradaG2 receive String | via saida1G2 send String |
            via saida2G2 send String | via saida3G2 send String)*
    }
    behavior is {
        via entradaG2 receive alerta : String
        compose{
            via saida1G2 send alerta
            and via saida2G2 send alerta
            and via saida3G2 send alerta
        }
    }
}
```

Figura 5.8 Descrição em π-ADL do componente Gateway2.

Componente Gateway8: a Figura 5.9 representa a descrição do componente Gateway8 que representa um gateway exclusivo, onde há uma entrada, representada por entradaG8 e duas saídas alternativas representada por saida1G8 e saida2G8.

```
component Gateway8 is abstraction (){
    connection entradaG8 is in(String)
    connection saida1G8 is out(String)
    connection saida2G8 is out(String)
    protocol is {
        (via entradaG8 receive String | via saida1G8 send String |
            via saida2G8 send String)*
    }
    behavior is {
        via entradaG8 receive relatorio2 : String
        if relatorio2=="controlado" then{
            via saida1G8 send relatorio2
            behavior()
        }
        else{
            via saida2G8 send relatorio2
            behavior()
        }
    }
}
```

Figura 5.9 Descrição em π-ADL do componente Gateway8.

Componente Gateway4: A Figura 5.10 representa a descrição do componente Ga-

teway4 que representa um gateway paralelo de junção, onde das duas entradas executadas ao mesmo tempo, representadas por entrada1G4 e entrada2G4 e apenas uma saída representada por saídaG4.

```
component Gateway4 is abstraction (){
    connection entrada1G4 is in(String)
    connection entrada2G4 is in(String)
    connection saidaG4 is out(String)
    protocol is {
        (via entrada1G4 receive String | via entrada2G4 receive String |
            via saidaG4 send String)*
    }
    behavior is {
        finalizacao is function(fim2: String) : String{
            return("")
        }
        compose{
        via entrada1G4 receive aviso : String
        and via entrada2G4 receive orientacao : String
        }
        via saidaG4 send finalizacao(aviso+orientacao)
        behavior()
    }
}
```

Figura 5.10 Descrição em π-ADL do componente Gateway4.

Componente Fim1: a Figura 5.11 mostra a especificação do componente Fim1 que é um evento de término, onde ocorre o final do processo, representado através da variável entrada, que ao ser recebida finaliza o processo.

```
component Fim is abstraction (){
    connection entrada is in(String)
    protocol is {
        (via entrada receive String)*
    }
    behavior is {
        via entrada receive servico : String
        done
    }
}
```

Figura 5.11 Descrição em π-ADL do componente Fim1.

5.1.3 Especificação das propriedades

Assim como é explicado na Subseção 4.3.3, devemos definir as propriedades a serem verificadas para analisar a presença ou não dos erros em tempo de execução. Além das já apresentadas na seção em questão, a seguir mostraremos mais algumas propriedades utilizadas nesse trabalho.

Garantir que todas as tarefas iniciadas enviem dados em menos de Y unidades de tempo. Assumindo que finalize conectando sua saída a entrada de um fluxo de sequência, tal declaração pode ser expressa da seguinte forma:

```
always during X time units {
  forall t:allOfType(Tarefa) {
    isTrue (t.entrada > 0) implies (eventually before Y time units {
    (exists fl:allOfType(Fluxo) areLinked (t.saida,fl.entrada))
    } )
  } }
```

Essa propriedade quando analisada for positiva garante ausência de deadlocks nas tarefas, caso contrário afirma a existência de erros.

A próxima propriedade faz referência a má execução de um gateway paralelo, nesta propriedade é exemplificada a ocorrência de deadlock:

```
always during X time units {
  forall g:allOfType(Gateway) {
    (exists fl:allOfType(Fluxo) areLinked(fl.para,g.entrada)) implies
    (eventually before Y time units {
      not(exists fl:allOfType(Fluxo) areLinked(g.saida1,fl.de)) and (exists
fl:allOfType(Fluxo) areLinked(g.saida2,fl.de))
    })
  } }
```

Ela indica que ao receber um valor de entrada através de um fluxo de sequência e após Y unidades de tempo o gateway não apresenta as duas saídas esperadas, isso indica a presença de um erro. Para a escrita dessa propriedade para um gateway exclusivo, basta alterar *and* da última função pelo *or*.

Essa próxima propriedade, também é referente a má execução de um gateway, mas desta vez se trata de um exclusivo de junção:

```
always during X time units {
  forall g:allOfType(Gateway2) {
    (exists fl:allOfType(Fluxo) areLinked(fl.para,g.entrada1)) or (exists
fl:allOfType(Fluxo6) areLinked(fl.para,g.entrada2)) implies
    (eventually before Y time units {
not(exists fl:allOfType(Fluxo7) areLinked(g.saida,fl.de))
    })
   } }
```

Neste caso indica que ao receber um dos dois valores de entrada através de fluxos de sequência e após Y unidades de tempo ele não apresenta a saída esperada, isso indica a presença de um deadlock. Assim como na propriedade anterior, para escrever a propriedade para um gateway paralelo basta mudar o *or* para primeira função por *and*.

5.1.4 Verificação do modelo e análise dos resultados

Após a execução das propriedades apresentadas acima obtemos os seguintes resultados:

- Primeiro apresentamos o resultado da verificação da propriedade referente as tarefas, que pode ser observado na Figura 5.12. Executando esta propriedade obtemos que em todas as simulações realizadas a propriedade é positiva, ou seja, a tarefa é executada da forma correta, livre de erros.

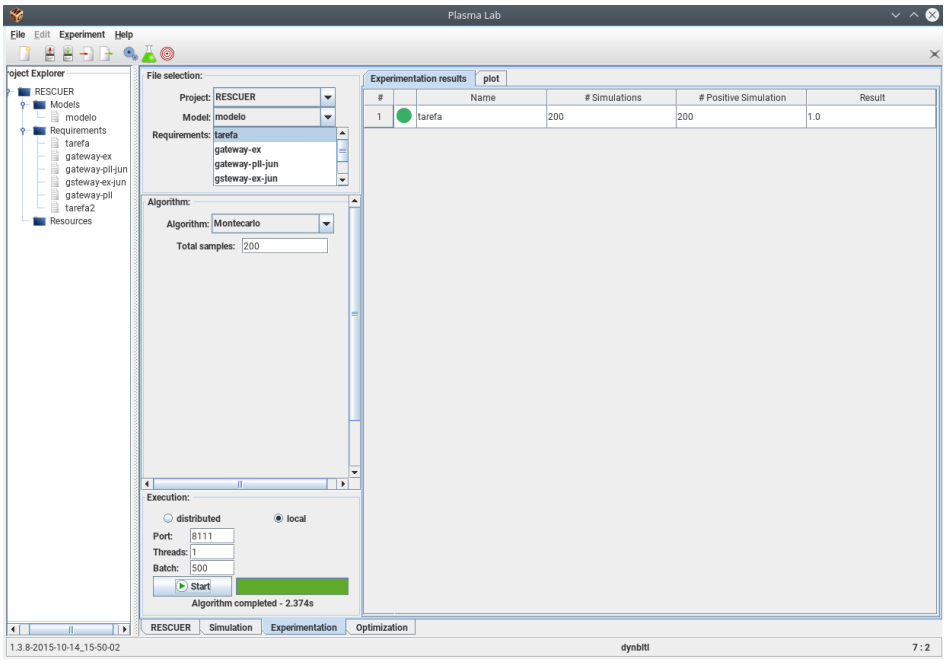


Figura 5.12 Verificação das tarefas.

- Em seguida realizamos a verificação da propriedade referente a gateways exclusivos, Figura 5.13:

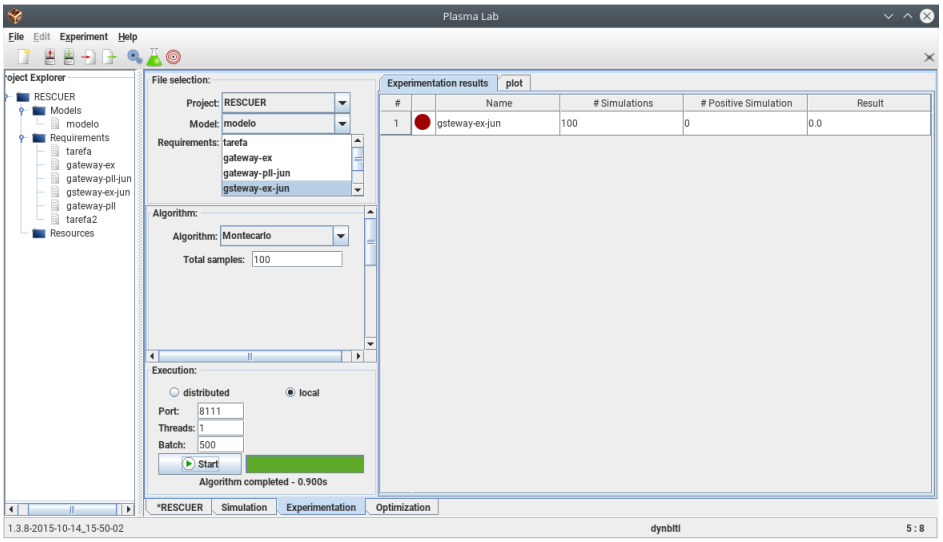


Figura 5.13 Verificação de gateways exclusivo.

Essa propriedade ao contrário da propriedade das tarefas, acusava presença de deadlock caso ela fosse positiva. Como se pode observar na figura o resultado da verificação não obteve simulações positivas, ou seja, o gateway verificado também está livre de erros.

- A próxima propriedade é referente a gateways paralelos, e obteve o resultado apresentado na Figura 5.14.

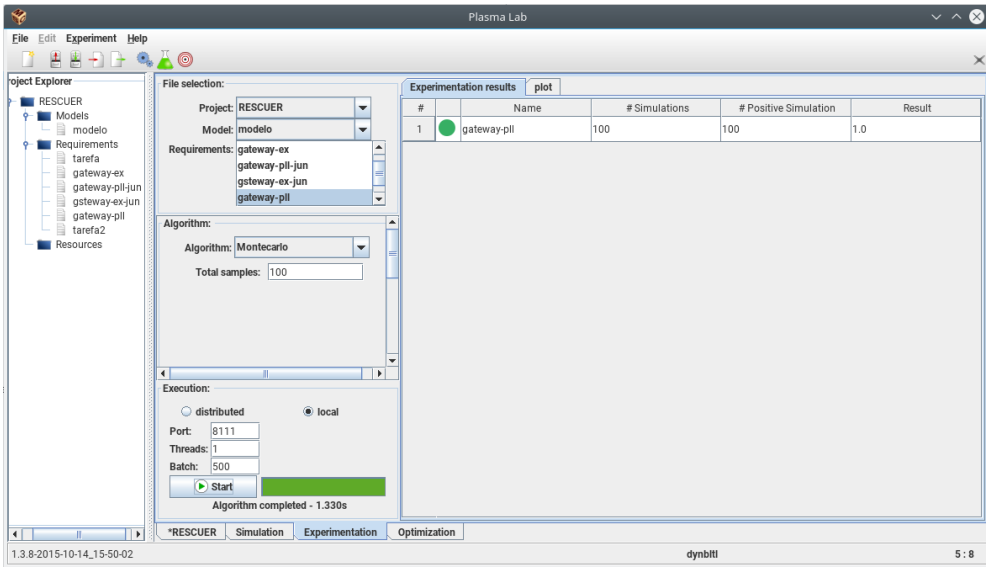


Figura 5.14 Verificação de gateways paralelo.

Assim como a propriedade anterior, esta também acusa presença de erro, mas desta vez apresentou durante a simulação, presença de simulações positivas, garantindo que o gateway em questão não realiza sua função corretamente.

Ao executar todas as propriedades apresentadas neste capítulo tivemos como o objetivo validar a nossa proposta e analisar se através dos passos realizados conseguimos detectar os possíveis erros em tempo de execução.

Através da verificação pudemos averiguar a existência ou não de erros em um modelo formal do sistema em um determinado espaço de tempo, conseguindo assim elucidar a questão referente a identificação da presença ou ausência de possíveis erros.

Porém, devido a limitação da ferramenta PLASMA, em casos da presença de erros, não temos como definir de qual localização do diagrama BPMN ele ocorre. Sendo assim, a nossa abordagem é capaz de detectar a presença de erros em tempo de execução, mas não possibilita sua correção.

Capítulo 6

CONCLUSÃO E CONSIDERAÇÕES FINAIS

Neste trabalho foi proposto um método para identificar erros que ocorrem em tempo de execução, como os deadlocks e livelocks, em Sistema de Sistemas modelados utilizando o diagrama de coreografia da notação BPMN, buscando evitar o mau funcionamento destes SoS. BPMN fornece um método para gerar modelos que permitem a uma organização entender seus procedimentos internos de negócios, em relação ao contexto de SoS, ele é usado para coordenar a sincronização de processos de negócios automatizados, ajudando a descrever os sistemas, produtos, serviços e sua arquitetura. Detectar erros em tempo de execução desses diagramas representa uma contribuição significativa.

Uma vez que em BPMN há uma falta de semântica formal, o π-ADL nos permite obter modelos formais a partir dos diagramada de coreografia, possibilitando sua verificação automática por meio de uma Verificação de Modelo Estatístico (SMC), que visa evitar o problema de explosão de estados, reduzindo o esforço o computacional e o tempo necessário para realizar a tarefa de verificação, que pode ocorrer ao usar verificações de modelo tradicionais.

Para isso, usamos um editor textual adequado para descrições de π-ADL e a ferramenta PLASMA para realizar a verificação das propriedades. Ao realizar as verificações, foi possível definir a presença de determinados erros ou a sua ausência, então dessa forma é possível garantir a precisão ou não de determinado diagrama. Infelizmente, não descobrimos como identificar o caminho que leva às falhas que a verificação pode revelar.

Desta forma, nosso trabalho conseguiu solucionar a questão referente a identificação da presença dos possíveis erros, mas não foi possível chegar na definição de em qual ponto do diagrama o erro ocorre, quando presente. Concluímos que nossa abordagem formal consegue detectar a presença de erros em tempo de execução dos modelos, porém sem a possibilidade de sua correção de forma automatizada.

Como trabalho futuro, tendo como base o conhecimento apresentado neste trabalho, já está em andamento a construção de uma ferramenta para automatizar o processo de tradução entre os diagramas coreografia BPMN e o modelo π -ADL correspondente e, tentando solucionar a questão da identificação do erro, que também permita analisar a possível presença de erros e informar a posição onde ele ocorre no diagrama de origem.

REFERÊNCIAS BIBLIOGRÁFICAS

AWAD, A.; PUHLMANN, F. Structural detection of deadlocks in business process models. In: SPRINGER. *International Conference on Business Information Systems*. [S.l.], 2008. p. 239–250.

BALDWIN, W. C.; SAUSER, B. Modeling the characteristics of system of systems. In: IEEE. *2009 IEEE International Conference on System of Systems Engineering (SoSE)*. [S.l.], 2009. p. 1–6.

BOARDMAN, J.; SAUSER, B. System of systems-the meaning of of. In: IEEE. *System of Systems Engineering, 2006 IEEE/SMC International Conference on*. [S.l.], 2006. p. 118–123.

BOYER, B. et al. Plasma-lab: A flexible, distributable statistical model checking library. In: SPRINGER. *International Conference on Quantitative Evaluation of Systems*. [S.l.], 2013. p. 160–164.

CAVALCANTE, E.; OQUENDO, F.; BATISTA, T. Architecture-based code generation: From π-adl architecture descriptions to implementations in the go language. In: SPRINGER. *European Conference on Software Architecture*. [S.l.], 2014. p. 130–145.

CAVALCANTE, E.; OQUENDO, F.; BATISTA, T. *π-ADL: a formal description language for software architectures*. [S.l.], 2014.

CAVALCANTE, E. et al. Statistical model checking of dynamic software architectures. In: SPRINGER. *European Conference on Software Architecture*. [S.l.], 2016. p. 185–200.

CAVALCANTE, E. R. d. S. A formally founded framework for dynamic software architectures. Brasil, 2016.

CBOK, B. Guide to the business process management common body of knowledge. *Version 3.0. 2009*, 2013.

FREUND, J.; RÜCKER, B. Real life bpmn. *Berlin: Camunda*, 2012.

HOLZMANN, G. J. The logic of bugs. *ACM SIGSOFT Software Engineering Notes*, ACM New York, NY, USA, v. 27, n. 6, p. 81–87, 2002.

HUTH, M.; RYAN, M. *Logic in Computer Science: Modelling and reasoning about systems*. [S.l.]: Cambridge university press, 2004.

JAMSHIDI, M. System of systems engineering-new challenges for the 21st century. *IEEE Aerospace and Electronic Systems Magazine*, IEEE, v. 23, n. 5, p. 4–19, 2008.

KARCANIAS, N.; HESSAMI, A. G. System of systems and emergence part 1: Principles and framework. In: IEEE. *Emerging Trends in Engineering and Technology (ICETET), 2011 4th International Conference on*. [S.l.], 2011. p. 27–32.

KHERBOUCHE, O. M.; AHMAD, A.; BASSON, H. Detecting structural errors in bpmn process models. In: IEEE. *Multitopic Conference (INMIC), 2012 15th International*. [S.l.], 2012. p. 425–431.

KHERBOUCHE, O. M.; AHMAD, A.; BASSON, H. Using model checking to control the structural errors in bpmn models. In: IEEE. *Research Challenges in Information Science (RCIS), 2013 IEEE Seventh International Conference on*. [S.l.], 2013. p. 1–12.

KIM, Y. et al. Validating software reliability early through statistical model checking. *IEEE software*, IEEE, p. 1, 2013.

LAUE, R.; AWAD, A. Visual suggestions for improvements in business process diagrams. *Journal of Visual Languages & Computing*, Elsevier, v. 22, n. 5, p. 385–399, 2011.

LEGAY, A.; DELAHAYE, B.; BENSALEM, S. Statistical model checking: An overview. In: SPRINGER. *International conference on runtime verification*. [S.l.], 2010. p. 122–135.

LUZEAUX, D.; RUAULT, J.-R. *Systems of systems*. [S.l.]: John Wiley & Sons, 2013.

MAIER, M. W. Architecting principles for systems-of-systems. In: WILEY ONLINE LIBRARY. *INCOSE International Symposium*. [S.l.], 1996. v. 6, n. 1, p. 565–573.

MEKDECI, B. et al. 5.2. 2 examining survivability of systems of systems. In: WILEY ONLINE LIBRARY. *INCOSE International Symposium*. [S.l.], 2011. v. 21, n. 1, p. 569–581.

OBJECT MANAGEMENT GROUP. *Business Process Model and Notation, V1.1*. [S.l.], 2008. Disponível em: <http://www.omg.org/spec/BPMN/1.1/PDF>.

OBJECT MANAGEMENT GROUP. *Business Process Model and Notation (BPMN) Version 2.0*. [S.l.], 2011. Disponível em: <http://www.omg.org/spec/BPMN/2.0>.

OQUENDO, F. π-adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, ACM, v. 29, n. 3, p. 1–14, 2004.

OQUENDO, F. Formal approach for the development of business processes in terms of service-oriented architectures using π-adl. In: IEEE. *Service-Oriented System Engineering, 2008. SOSE'08. IEEE International Symposium on*. [S.l.], 2008. p. 154–159.

OQUENDO, F. π-adl for ws-composition: A service-oriented architecture description language for the formal development of dynamic web service compositions. In: *SBCARS*. [S.l.: s.n.], 2008. p. 52–66.

SAGE, A. P.; CUPPAN, C. D. On the systems engineering and management of systems of systems and federations of systems. *Information knowledge systems management*, IOS Press, v. 2, n. 4, p. 325–345, 2001.

TANTITHARANUKUL, N.; SUGUNNASIL, P.; JUMPAMULE, W. Detecting deadlock and multiple termination in bpmn model using process automata. In: IEEE. *Electrical Engineering/Electronics Computer Telecommunications and Information Technology (ECTI-CON), 2010 International Conference on*. [S.l.], 2010. p. 478–482.

VILLELA, K. et al. Reliable and smart decision support system for emergency management based on crowdsourcing information. In: *Exploring Intelligent Decision Support Systems*. [S.l.]: Springer, 2018. p. 177–198.

XIAO, B. et al. Reasearch on the history and perspective of system of systems. In: IEEE. *Reliability, Maintainability and Safety (ICRMS), 2011 9th International Conference on*. [S.l.], 2011. p. 1262–1266.

XU, H. et al. Business process modeling and design of smart home service system. In: IEEE. *2012 International Joint Conference on Service Sciences*. [S.l.], 2012. p. 12–17.