
UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Paulo Henrique Nascimento Cavalcanti

# Verificação de Modelos Comportamentais UML Como um Serviço Habilitando a Aplicação de Métodos Formais Ocultos

Recife

2022

Paulo Henrique Nascimento Cavalcanti

# Verificação de Modelos Comportamentais UML Como um Serviço Habilitando a Aplicação de Métodos Formais Ocultos

Monografia apresentada ao Curso de Bacharelado em Ciências da Computação da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Ciências da Computação.

Universidade Federal Rural de Pernambuco – UFRPE

Departamento de Computação

Curso de Bacharelado em Ciências da Computação

Orientador: Lucas Albertins de Lima

Recife

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal Rural de Pernambuco
Sistema Integrado de Bibliotecas
Gerada automaticamente, mediante os dados fornecidos pelo(a) autor(a)

C377v Cavalcanti, Paulo Henrique Nascimento
Verificação de Modelos Comportamentais UML Como um Serviço Habilitando a Aplicação de Métodos Formais Ocultos / Paulo Henrique Nascimento Cavalcanti. - 2022.
48 f. : il.

Orientador: Lucas Albertins de Lima.
Inclui referências.

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal Rural de Pernambuco, Bacharelado em Ciência da Computação, Recife, 2022.

1. UML. 2. diagrama de atividade. 3. diagrama de máquina de estados. 4. verificação. 5. microsserviço. I. Lima, Lucas Albertins de, orient. II. Título

CDD 004

**MINISTÉRIO DA EDUCAÇÃO E DO DESPORTO**
**UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO (UFRPE)**
**BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**
**http://www.bcc.ufrpe.br**

## FICHA DE APROVAÇÃO DO TRABALHO DE CONCLUSÃO DE CURSO

Trabalho defendido por Paulo Henrique Nascimento Cavalcanti às 08 horas do dia 04 de outubro de 2022, no link https://meet.google.com/fad-xomv-wgp, como requisito para conclusão do curso de Bacharelado em Ciência da Computação da Universidade Federal Rural de Pernambuco, intitulado "Verificação de Modelos Comportamentais UML Como um Serviço Habilitando a Aplicação de Métodos Formais Ocultos", orientado por Lucas Albertins de Lima e aprovado pela seguinte banca examinadora:

---

Lucas Albertins de Lima

DC/UFRPE

---

Sidney Nogueira

DC/UFRPE

*A todos aqueles que estiveram presentes durante esta jornada.*

# Agradecimentos

Agradeço, em primeiro lugar, aos meus pais, por terem sempre me incentivado a buscar educação, por terem sempre me colocado em primeiro lugar e me proporcionado uma vida tranquila, na qual pude correr atrás dos meus sonhos. Jamais teria chegado até aqui sem o apoio deles e de toda a família.

Agradeço também ao meu orientador por ter aceitado me acompanhar nesta última etapa do curso e por ter lecionado algumas das minhas disciplinas favoritas deste curso com maestria.

Por fim, agradeço aos amigos, que tornaram a caminhada até aqui mais suave e auxiliaram nos mais diversos assuntos inúmeras vezes.

*“Um único sonho é mais poderoso do que mil realidades.”*
*(J.R.R Tolkien)*

# Resumo

Conforme os sistemas vão ficando mais complexos, mais esforço tem sido necessário para realizar validações sobre eles. Além da complexidade, o custo de correções também aumenta conforme os projetos avançam de fase, tornando essencial a detecção de erros nos estágios mais iniciais. Dentro da *Model-Based Systems Engineering*, a verificação de modelos é uma das possíveis abordagens para a solução do problema. Contudo, realizar tal verificação envolve muitas vezes a utilização de métodos formais. Estes métodos são complexos e nem todos os projetistas de sistemas têm conhecimento deles. Um outro ponto importante é que, dada a necessidade cada vez maior de suportar grandes cargas, é comum a utilização de concorrência nos sistemas. Essa natureza concorrente dos sistemas atuais, traz consigo a possibilidade da inclusão de problemas como *deadlock* e não-determinismo, normalmente não verificados pelas ferramentas atuais, que muitas vezes também exigem licenças para sua utilização e oferecem poucas possibilidades de integração com outras ferramentas ou ambientes. Neste sentido, nosso trabalho utiliza-se do arcabouço construído em trabalhos anteriores parwa expandir a possibilidade da checagem de modelos através de microsserviços, gratuitos e de código aberto. Apesar de outros trabalhos já realizarem a verificação de propriedades em modelos UML, em sua maioria, eles dependem da instalação de ferramentas e permitem pouca ou nenhuma integração com outros sistemas. Sendo assim, nossa principal contribuição é a construção de uma arquitetura baseada em microsserviços para disponibilizar serviços de verificação de propriedades clássicas (deadlock e não-determinismo) para um subconjunto de modelos UML comportamentais, mais precisamente diagramas de atividade e de máquinas de estado.

**Palavras-chave**: UML, diagrama de atividade, diagrama de máquina de estados, verificação, microsserviço.

# Abstract

As systems become more complex, more effort is required to perform validations on them. In addition to complexity, the cost of corrections also increases as projects progress, making error detection at the earliest stages essential. Within Model-Based Systems Engineering, model verification is one of the possible approaches to solving the problem. However, performing such verification often involves the use of formal methods. These methods are complex and not all system designers are knowledgeable in them. Another important point is that, given the increasing need to support large loads, it is common to use concurrency in systems. This concurrent nature of systems brings with it the possibility of including problems such as deadlock and non-determinism, usually not verified by current tools, which often also require licenses for their use and offer little possibilities of integration with other tools and environments. In this sense, our work uses the framework built in previous initiatives to expand the possibility of checking models through free and open source microservices. Although other works have already performed the verification of properties in UML models, most of them depend on the installation of tools and allow little or no integration with other systems. Therefore, our main contribution is the construction of a microservices-based architecture to provide services for checking classic properties (deadlock and non-determinism) for a subset of behavioral UML models, more precisely activity diagrams and state machines.

**Keywords**: UML, activity diagram, machine state diagram, verification, microservice.

# Lista de ilustrações


Figura 1 – Custo para correção de erros de acordo com a fase do projeto baseado em . . . 13
Figura 2 – Exemplo de um sistema de reservas de hotel representado em um diagrama de atividade . . . 14
Figura 3 – Aspirador de pó representado em um diagrama de máquina de estados 14
Figura 4 – Hierarquia dos tipos de diagramas . . . 20
Figura 5 – Exemplo de um diagrama de atividades simples . . . 21
Figura 6 – Diferentes tipos de nós . . . 22
Figura 7 – Diferentes tipos de estados . . . 23
Figura 8 – Exemplo de uma arquitetura em microsserviços . . . 26
Figura 9 – Visão Geral do Serviço . . . 27
Figura 10 – Front-end criado para a validação de diagramas. . . . 28
Figura 11 – Diversas formas de utilização do Serviço . . . 29
Figura 12 – Arquitetura do microsserviço em detalhes . . . 30
Figura 13 – Contratos do Serviço . . . 31
Figura 14 – Retorno do endpoint /index . . . 32
Figura 15 – Retorno do endpoint /getDiagrams . . . 32
Figura 16 – Retorno do endpoint /validateAstahFile . . . 33
Figura 17 – Diagrama validado . . . 33
Figura 18 – Utilização do front-end padrão do serviço . . . 34
Figura 19 – Diagrama de um sistema de comércio online . . . 35
Figura 20 – Diagrama do sistema de e-commerce após verificação de deadlock 36
Figura 21 – Diagrama do sistema de e-commerce após verificação de deadlock 36
Figura 22 – Diagrama de um sistema de armazenamento de arquivos . . . 37
Figura 23 – Resultado obtido após executar a requisição via Swagger . . . 37
Figura 24 – Diagrama de um sistema de armazenamento de arquivos após verificação de não-determinismo . . . 38
Figura 25 – Interface do Postman com requisição montada para validar um diagrama 39
Figura 26 – Diagrama de um sistema de classificação de contas bancária . . . 39
Figura 27 – Resposta da requisição feita pelo Postman . . . 40
Figura 28 – Diagrama de um sistema de classificação de contas bancária após validação de não-determinismo . . . 41
Figura 29 – Aplicação *desktop* para validação de diagramas . . . 42
Figura 30 – Diagrama de um sistema de gerenciamento de contratos . . . 43
Figura 31 – Opção de salvar o arquivo validado dentro do sistema . . . 44

Figura 32 – Diagrama de um sistema de gerenciamento de contratos após verificação de deadlock . . . . . 45

# Lista de tabelas

# Lista de abreviaturas e siglas

| | |
|---|---|
| UML | Unified Modeling Language |
| CSP | Communicating Sequential Processes |
| FDR | Failures-Divergences Refinement |
| MBSE | Model-Based Systems Engineering |
| ESA | European Space Agency |
| OMG | Object Management Group |
| HTTP | Hypertext Transfer Protocol |
| API | Application Programming Interface |
| AJAX | Asynchronous JavaScript and XML |
| HTML | HyperText Markup Language |
| CSS | Cascading Style Sheets |
| JVM | Java Virtual Machine |
| REST | Representational State Transfer |

# Sumário


**Lista de ilustrações** . . . . . 7

**1 INTRODUÇÃO** . . . . . 12
**1.1 Objetivos** . . . . . 16
**1.2 Metodologia** . . . . . 17
**1.3 Organização do trabalho** . . . . . 18

**2 REFERENCIAL TEÓRICO** . . . . . 19
**2.1 Unified Modeling Language** . . . . . 19
2.1.1 Diagramas de Atividade . . . . . 21
2.1.2 Diagramas de Máquina de Estado . . . . . 22
**2.2 Model-Based System Engineering** . . . . . 23
2.2.1 Verificação de Modelos (*Model Checking*) . . . . . 24
**2.3 Microsserviços** . . . . . 25

**3 SERVIÇO DE VERIFICAÇÃO DE MODELOS COMPORTAMENTAIS UML** . . . . . 27
**3.1 Visão Geral da Arquitetura dos Serviços** . . . . . 27
3.1.1 Tecnologias Utilizadas na Implementação . . . . . 29
**3.2 Estrutura detalhada dos microsserviços** . . . . . 30
**3.3 Contrato dos Serviços** . . . . . 31

**4 AVALIAÇÃO** . . . . . 34
**4.1 Utilizando Front-end Provido pelo Serviço** . . . . . 34
**4.2 Utilizando Swagger** . . . . . 36
**4.3 Utilizando Postman** . . . . . 38
**4.4 Utilizando Aplicação Desktop** . . . . . 41

**5 CONCLUSÃO** . . . . . 46
**5.1 Trabalhos Relacionados** . . . . . 46
**5.2 Limitações e Trabalhos Futuros** . . . . . 47

**REFERÊNCIAS** . . . . . 48

# 1 Introdução

Em junho de 1996, aconteceu o primeiro lançamento do foguete Ariane 5, da *European Space agency* (ESA). Cerca de 40 segundos após o lançamento, o foguete desviou da sua rota prevista e explodiu. Após análise da ESA, foi publicado o relatório (LIONS, 1996), no qual ficou constatada a causa do desastre: perda total de informações de altitude e orientação. Esta perda de informações, por sua vez, foi causada por erros de *design* e especificação no *software* do foguete, que não foram detectados em sua bateria de testes.

De acordo com (HORVÁTH et al., 2020), é possível explorar somente algumas interações típicas e calcular valores comuns para propriedades dos sistemas através de testes comuns e simulações. Dessa forma, quando o sistema modelado for muito extenso e/ou complexo, como no caso do Ariane 5, falhas sutis podem ocorrer sem que as baterias de testes os detectem.

Um modelo, conforme (SHEVCHENKO, 2020), é uma representação simplificada de algo. É uma representação utilizada para eliminar complexidade, seja matemática, gráfica ou física. Tais modelos devem ser suficientes para representar todo o sistema e o sistema deve confirmá-los. Ainda segundo (HORVÁTH et al., 2020), a *Model-Based Systems Engineering* (MBSE), coloca modelos em primeiro plano durante todas as atividades de engenharia. Desde o *design* até a validação, além de seu papel original, que é o de documentação.

Comparados aos métodos tradicionais de testes, como testes unitários, os métodos formais utilizam mecanismos que nos trazem uma maior assertividade em relação às possíveis falhas de um sistema. (CLARKE et al., 2018) define a Verificação de Modelos (*Model Checking*) como sendo um método computacional para a análise de sistemas. Este método é capaz de verificar todos os caminhos dentro de um modelo, mostrando assim, todos os pontos passíveis de comportamentos inesperados.

Tendo modelos como protagonistas, é possível iniciar a validação de problemas desde a fase de *design* do projeto, onde é mais barato corrigir falhas. (HASKINS et al., 2004) mostra em seu trabalho os fatores de custo para a correção de problemas de acordo com cada fase do projeto e, conforme o projeto avança de fase, mais caras ficam as correções, podendo facilmente fazer com que os orçamentos sejam extrapolados, como pode ser visto na Figura 1. Unindo isto ao fato da complexidade crescente dos sistemas atuais, é possível perceber a necessidade de detectar falhas ainda cedo nos cronogramas dos projetos.

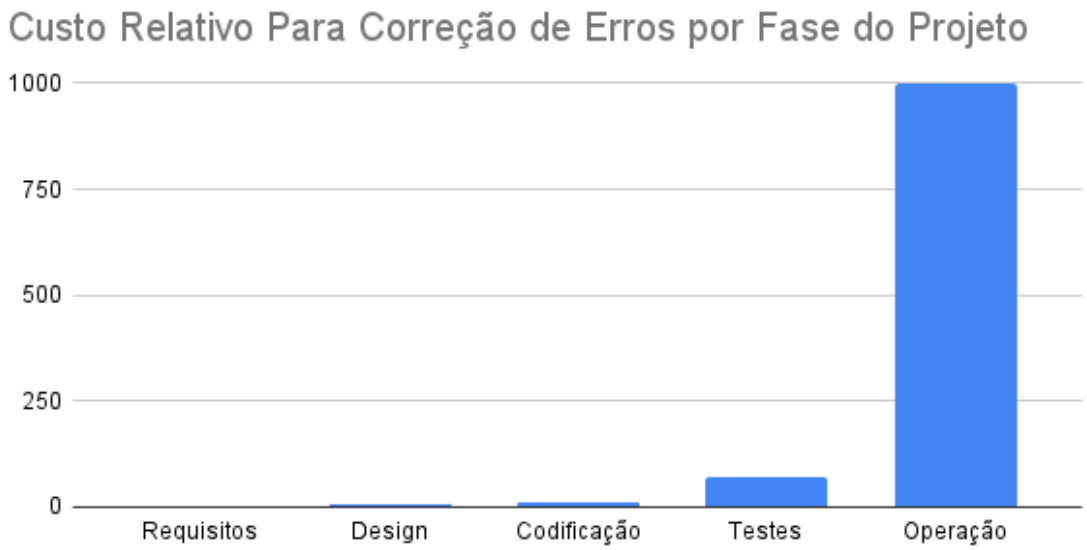


Figura 1 – Custo para correção de erros de acordo com a fase do projeto baseado em (HASKINS et al., 2004)

A verificação (ou checagem) de modelos é uma abordagem bem estabelecida dentro da utilização de métodos formais. De acordo com (CLARKE; WING, 1996) esta é uma técnica que consiste em construir um modelo finito, sobre o qual é realizada uma busca exaustiva de estado, garantindo que determinada propriedade é verdadeira para o modelo sendo analisado.

Apesar das vantagens de proporcionar verificações rigorosas em sistemas, o uso de métodos formais não tem se difundido devido à complexidade em manipular notações matemáticas. Enquanto isso, linguagens diagramáticas possuem um apelo visual e uma maior facilidade de especificação e compreensão, quando comparadas às linguagens formais. Dessa forma, diversos trabalhos buscam unir o melhor dos dois mundos, através do uso de métodos formais, de maneira oculta, como uma camada abaixo do nível diagramático, para permitir a verificação destes modelos enquanto se mantém a usabilidade no nível diagramático. (VISSER; DWYER; WHALEN, 2012) nomeia este tipo de abordagem de métodos formais ocultos (*hidden formal methods*).

Um bom exemplo de linguagem diagramática é a *Unified Modeling Language* (UML). Esta linguagem é utilizada principalmente para *design* de sistemas, conforme mostrado em (KOC et al., 2021), onde quase 70% dos casos de uso da linguagem foram para este princípio. Ainda segundo (KOC et al., 2021), dentro da UML, o diagrama de atividades é o segundo mais popular, perdendo somente para os diagramas de classe.

Estes diagramas de atividades são utilizados para fornecer uma notação gráfica que define a composição sequencial, condicional e paralela de comportamentos de níveis mais baixos, como diz (BARESI, 2009). Normalmente estes diagramas são utilizados para representar casos de uso mais simples, ou detalhar alguma lógica de

negócio. Porém, nada os impede de serem utilizados para representar lógicas mais complexas ou até mesmo sistemas inteiros. Na Figura 2 nós mostramos um exemplo de um fluxo de trabalho de um sistema de reserva de hotel que utiliza esta notação.

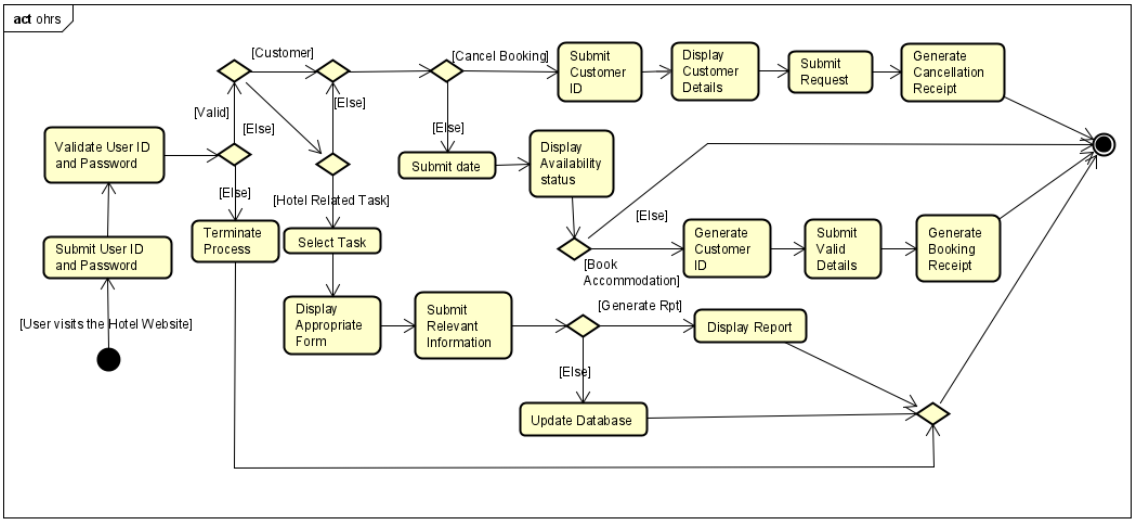


Figura 2 – Exemplo de um sistema de reservas de hotel representado em um diagrama de atividade

Além dos diagramas de atividade, existem os de máquina de estados, que são bastante utilizados para representar os possíveis estados de um objeto em um dado momento no tempo. Na Figura 3 é mostrado um diagrama de máquina de estados que representa um aspirador de pó.

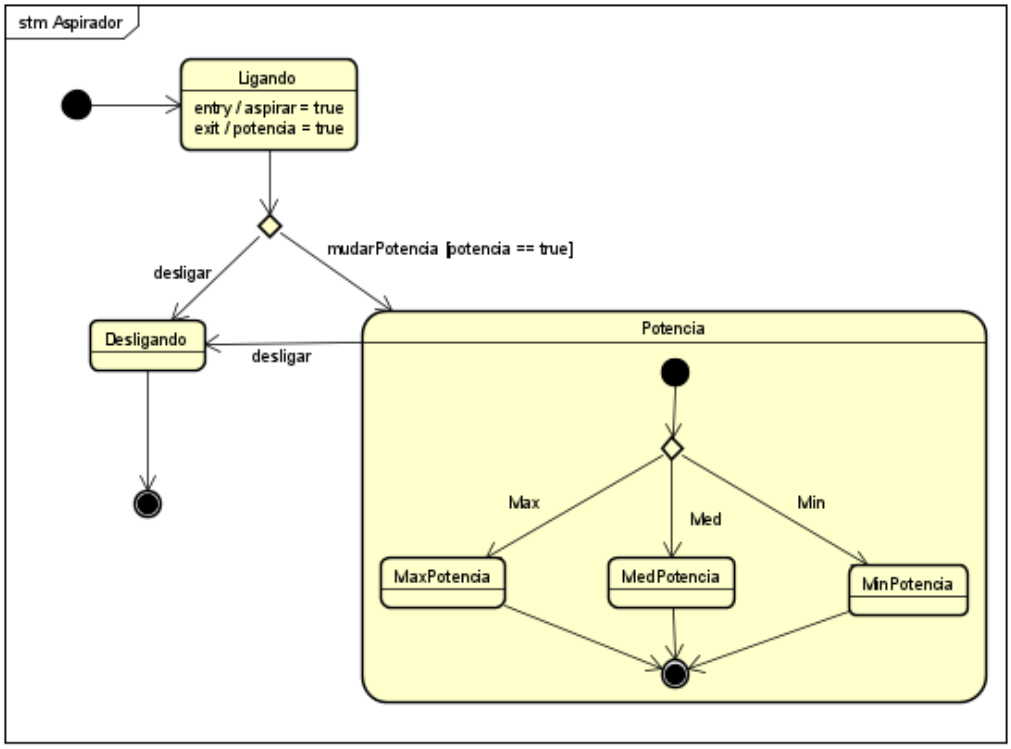


Figura 3 – Aspirador de pó representado em um diagrama de máquina de estados

Apesar de ter um apelo visual maior e também grande facilidade de entendimento, o uso de UML para construção de modelos introduz alguns problemas. Um destes é a necessidade de realizar as verificações de modelos de forma manual, o que em muitos dos casos acaba introduzindo falhas humanas nos projetos. Os verificadores de modelo, por sua vez, têm a capacidade de realizar estas verificações de maneira automática, exaustiva e livre de erros.

Trabalhos anteriores, como o de (LIMA; TAVARES; NOGUEIRA, 2020) mostram alguns dos erros mais comuns nestes modelos. Dentre eles, estão presentes os de não-determinismo e *deadlock*.

As definições de não-determinismo são antigas e muitas vezes relacionadas de maneira próxima ao conceito de concorrência. (ROSCOE, 2005) diz que o não determinismo acontece quando a partir de uma entrada específica, é possível obter mais de uma saída. No caso de um diagrama de atividades, por exemplo, isso seria refletido quando a partir de um nó é possível partir para dois ou mais nós sem definições claras de como se deve escolher dentre as diferentes direções.

Já o *deadlock*, conforme (ANTHONY; MARSLAND, 1980), acontece quando membros de um grupo de processos que detêm recursos são bloqueados indefinidamente do acesso a estes recursos, devido a outros processos do grupo estarem os acessando, impedindo que qualquer progresso seja feito. Essa situação pode ser exemplificada da seguinte forma: considerando uma situação em que quatro carros encontram-se em um cruzamento, cada carro em uma das partes. Para que não aconteça nenhum acidente, é necessário que o primeiro carro ceda a passagem para o segundo e assim por diante. Ou seja, para o primeiro carro, a via pertence (no momento) ao segundo e assim em diante, fazendo com que nenhum deles possa avançar. Nessa situação, os carros podem ser considerados os processos e a rua é o recurso.

Diversos trabalhos anteriores buscam automatizar a verificação de propriedades de modelos. A maior parte deles, como (HORVÁTH et al., 2020) e (OUCHANI; MOHAMED; DEBBABI, 2014) utiliza a abordagem de traduzir os modelos para uma linguagem intermediária, na qual é possível realizar as verificações de propriedades. Um outro trabalho, (LIMA; TAVARES; NOGUEIRA, 2020), propõe o uso da linguagem CSP para verificação de propriedades. Após a tradução do modelo para a linguagem, é utilizada uma ferramenta, chamada FDR, para a validação das propriedades do modelo. A ferramenta faz esta validação e retorna, caso haja problemas, os locais onde estão presentes.

O principal problema destas abordagens é que estão vinculadas a alguma plataforma específica. No caso de (LIMA; TAVARES; NOGUEIRA, 2020), o trabalho foi construído baseado totalmente na ferramenta de modelagem Astah (VISION, 2022). Já em (HORVÁTH et al., 2020) e (OUCHANI; MOHAMED; DEBBABI, 2014), é necessário

que os modelos estejam escritos em linguagem SysML. Um outro problema é o de que ferramentas de modelagem, como Visual Paradigm, StarUML, assim como outras, necessitam de licenças para utilização, o que torna difícil o acesso a ferramentas de validação de modelos.

Além destes, é possível citar mais um problema: para permitir interoperabilidade destes modelos os usuários devem fazer uso de versões locais destas ferramentas para adquirir artefatos. Isto dificulta a comunicação com outros sistemas.

Uma tendência criada por grandes players como Amazon, Netflix e Spotify é a de migração de sistemas monolíticos para microsserviços (AUER et al., 2021). Esta nova tendência é uma abordagem para construção de sistemas utilizando um conjunto de pequenos serviços autônomos que se comunicam de forma leve, normalmente utilizando o protocolo HTTP (LEWIS; FOWLER, 2014).

A arquitetura de microsserviços é altamente descentralizada e traz consigo diversos benefícios como alta escalabilidade, ciclos de deploy mais rápidos e boa separação de responsabilidades dentre os serviços (BUSHONG et al., 2021). Dessa forma, é possível atingir vários usuários de maneira simultânea e ainda assim garantir uma boa qualidade dos serviços.

Nosso trabalho irá focar no desenvolvimento de microsserviços focados na validação de modelos UML. A possibilidade de checagem em múltiplas plataformas e sem a necessidade de uma instalação específica proporcionará uma menor necessidade de conhecimento específico do usuário e o uso de uma arquitetura de microsserviços permitirá validações rápidas, podendo talvez evitar erros catastróficos como o de (LIONS, 1996).

## 1.1 Objetivos

Nesse trabalho iremos continuar o que foi desenvolvido em (LIMA; TAVARES; NOGUEIRA, 2020), extendendo a utilização para um ambiente de microsserviços. Nossa motivação é transformar o que foi construído para uso exclusivo em desktops, na plataforma Astah (VISION, 2022) em serviços que permitem não somente facilidade de uso por usuários, como também integração com outras ferramentas e serviços.

**Objetivo Geral**

Desenvolver um conjunto de microsserviços que permitam a validação de modelos comportamentais (atividades e máquina de estados) UML utilizando mais de uma plataforma.

**Objetivos Específicos**

1. Projetar uma arquitetura baseada em microsserviços para as verificações dos modelos UML no arcabouço construído na ferramenta Astah para verificação de modelos UML comportamentais.
2. Construir microsserviços para a validação de deadlock e não-determinismo para diagramas de atividades suportados na plataforma Astah.
3. Construir microsserviços para a validação de deadlock e não-determinismo para diagramas de máquina de estados suportados na plataforma Astah.
4. Desenvolver um ponto focal para estes serviços serem acessados, como uma API Gateway ou aplicação front-end.

## 1.2 Metodologia

A seguir nesta seção estão descritas as etapas de pesquisa contemplando todos os requisitos de modelagem, desenvolvimento e validação da solução proposta.

1. Levantar dados e trabalhos relacionados, validar proposta através da avaliação de soluções computacionais existentes para a verificação de modelos UML.
2. Definir e avaliar requisitos dos microsserviços, como as funcionalidades e tecnologias a serem utilizadas.
3. Modelar entidades e requisitos funcionais do sistema, como modelos de dados e seus relacionamentos, mapear casos de uso e fluxos de navegação do usuário.
4. Implementar requisitos previamente modelados, visando obter um conjunto de serviços que permita a entrada de diagramas UML e produza uma saída válida, com os erros encontrados no diagrama.
5. Utilizar diagramas reais e validar que as saídas produzidas são válidas.
6. Sintetizar o resultado obtido em forma de monografia, analisar pontos de melhoria e problemas encontrados durante a pesquisa e defesa do trabalho de conclusão de curso.

## 1.3 Organização do trabalho

Nosso trabalho está organizado em 5 capítulos, descritos a seguir:

- O Capítulo 2 descreve nosso referencial teórico, mostrando com mais profundidade os conceitos necessários para o desenvolvimento do trabalho.
- O Capítulo 3 apresenta efetivamente os serviços construídos para verificar propriedades de diagramas de atividade e máquinas de estado, mostrando seus detalhes.
- O Capítulo 4 apresenta a avaliação de resultados obtidos no trabalho através do uso dos microsserviços de diversos ponto de vista.
- Por fim, o Capítulo 5 apresenta as considerações finais e possíveis trabalhos futuros.

# 2 Referencial Teórico

Nesse capítulo iremos apresentar, em mais detalhes, os conceitos necessários para o entendimento do trabalho. Na Seção 2.1 introduzimos conceitos básicos sobre UML e os diagramas envolvidos neste trabalho, os quais são Diagramas de Atividades e Diagramas de Máquina de Estados. Na Seção 2.2 discutimos conceitos relacionados a *Model-based System Engineering*, e, por último, a Seção 2.3 descreve tópicos relacionados a Microsserviços.

## 2.1 Unified Modeling Language

Nesta seção iremos tratar da *Unified Modeling Language* (UML). A Subseção 2.1.1 irá trazer mais detalhes sobre os diagramas de atividade, enquanto a Subseção 2.1.2 apresenta o diagrama de máquinas de estado.

De acordo com (FOWLER, 2003), a *Unified Modeling Language*, nada mais é do que uma família de notações gráficas. Tal família é apoiada por um único meta-modelo, que a auxilia a descrever e projetar diversos sistemas, em especial aqueles construídos utilizando a orientação a objetos. Ainda de acordo com (FOWLER, 2003), a motivação por trás do uso de UML é a de que as linguagens de programação tradicionais não estão prontas para exibir um alto nível de abstração. Este nível é facilmente provido por linguagens como a UML e auxilia grandemente em discussões a respeito do design de sistemas.

A UML é um padrão aberto, controlado pela *Object Management Group* (OMG), um grupo criado com o objetivo de construir padrões que suportam a interoperabilidade de sistemas.

Os usos da linguagem são variados, mas em seu princípio, o objetivo era o de modelagem de *software*. Atualmente, porém, a UML tornou-se uma linguagem de propósito geral para modelagem, tendo diversas extensões criadas para utilizações específicas, como SysML, para modelagem de sistemas e SoaML, para modelagem de serviços dentro de uma arquitetura orientada a serviços. Dentro desse primeiro uso, (FOWLER, 2003) nos mostra três utilizações: rascunho, planejamento e linguagem de programação. Explicamos estes usos abaixo:

- Rascunho: uma utilização um tanto informal da linguagem, os rascunhos podem ser feitos para discutir a ideia de como um sistema funcionará com um grupo de pessoas.

- Planejamento: este é um uso mais formal da linguagem. Aqui, é possível utilizar de artifícios como diagramas de classe para especificar elementos que existirão no projeto de software futuramente, mantendo-os documentados.
- Por fim, o uso como linguagem de programação: este é um caso de uso menos comum, porém, com modelos suficientemente bem descritos em UML, é possível utilizar ferramentas de geração de código, que traduzirão as especificações UML para alguma linguagem de programação.

Para o segundo e terceiro caso de uso é muito importante que os modelos construídos estejam verificados e validados, evitando que o sistema gerado futuramente esteja com erros e se comporte como o esperado.

Dentro destes usos existem, oficialmente, 14 tipos de diagramas que estão divididos em duas categorias: estruturais e comportamentais. Esta última tem uma subcategoria chamada de diagramas interativos ou de interatividade. A Figura 4 mostra todos estes diagramas dentro de sua hierarquia.

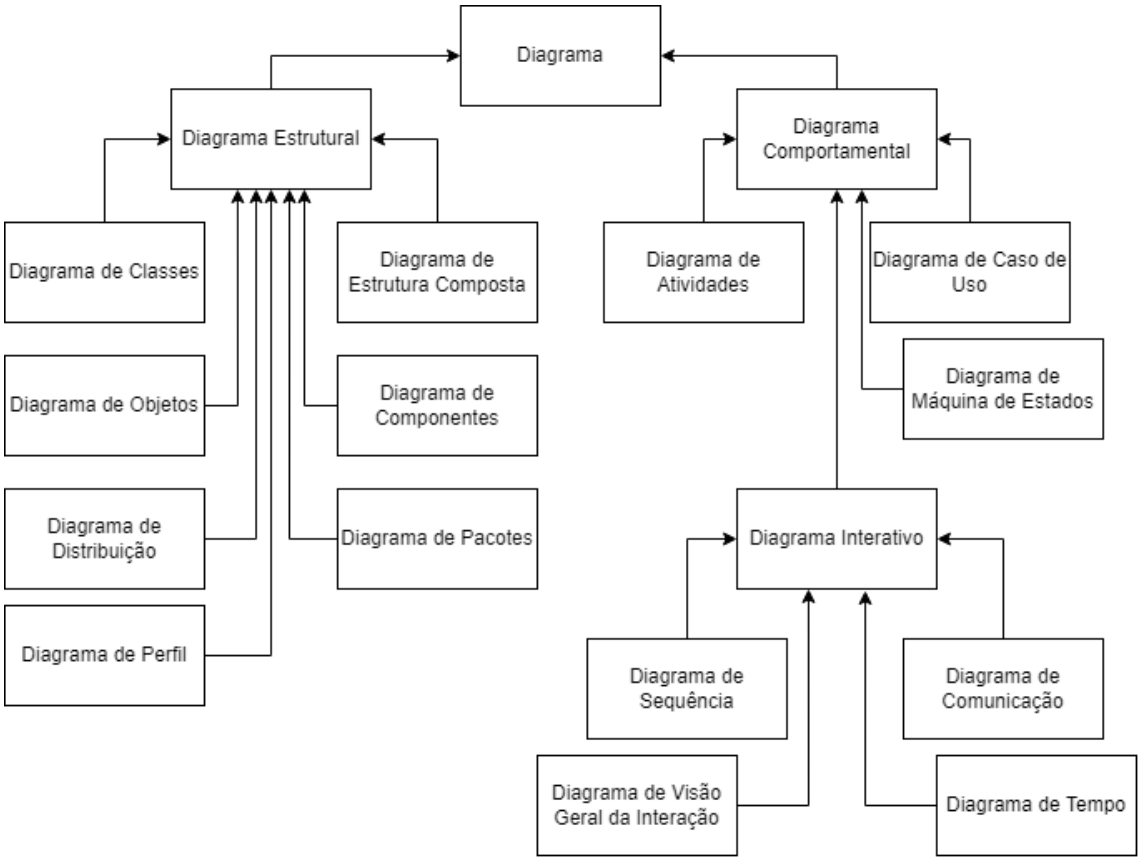


Figura 4 – Hierarquia dos tipos de diagramas
baseada em (GROUP, 2017)

Neste trabalho utilizaremos o diagrama de atividades e o de máquina de estado, explicados nas subseções 2.1.1 e 2.1.2 respectivamente.

### 2.1.1 Diagramas de Atividade

O uso de diagramas de atividade está relacionado com a necessidade de descrever como as atividades estão relacionadas para prover um serviço (PARADIGM, 2022). Atualmente este é um dos tipos de diagrama mais populares, sendo o segundo mais utilizado na literatura, segundo (KOC et al., 2021).

Uma atividade, seguindo a especificação de (GROUP, 2017), é um tipo de comportamento especificado como um grafo, que tem nós interconectados por arestas. Um subconjunto de tais nós é o de “nós de ação”, que corresponde a outras atividades em um nível inferior ao atual, ou a execução de algum comportamento. Outro subconjunto é o de “nós de objeto”, que são responsáveis por guardar valores de entradas e saídas do subconjunto anterior. Além destes, existem os “nós de controle” que especificam o sequenciamento de nós por meio de arestas de fluxo de controle.

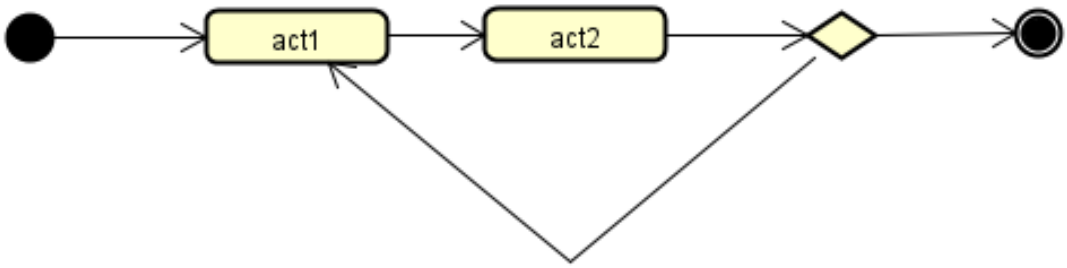


Figura 5 – Exemplo de um diagrama de atividades simples

Os nós executáveis (ou de ação), são responsáveis por realizar uma etapa comportamental considerável da atividade. Todas as arestas de entrada e saída deste tipo de nó devem ser fluxos de controle. Para consumir e produzir dados é necessário utilizar nós de objetos. Na Figura 5, os nós com nome “act1” e “act2” são de ação.

Já os nós de controle são utilizados para gerenciar o fluxo de valores entre os demais nós da atividade. No total, existem sete tipos diferentes de nós de controle: nós iniciais, nós de final de fluxo, nós de final de atividade, nós de bifurcação, nós de junção, nós de mesclagem e nós de decisão. Explicamos abaixo cada um destes tipos, com a visualização de cada um desses nós na Figura 6:

- Nós iniciais: Iniciam a execução do fluxo.
- Nós de final de fluxo: Finalizam a execução do fluxo.
- Nós de final de atividade: Finalizam a execução da atividade.
- Nós de bifurcação: Separam o fluxo em múltiplos fluxos concorrentes.

- Nós de junção: Mesclam fluxos separados em um único, com sincronização.
- Nós de mesclagem: Mesclam fluxos separados em um único, sem sincronização.
- Nós de decisão: Selecionam, através de guardas, qual fluxo deve ser seguido.

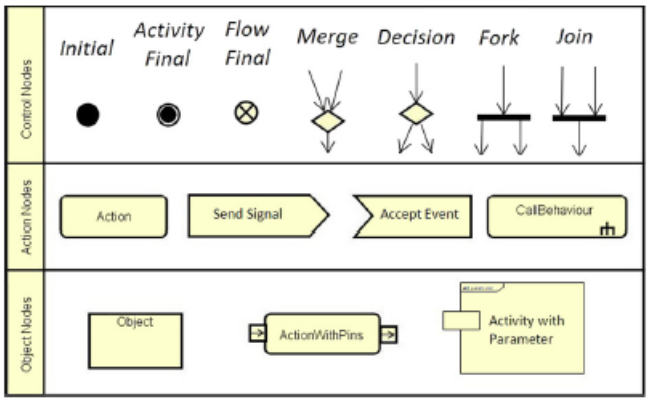


Figura 6 – Diferentes tipos de nós
retirada de (LIMA; TAVARES; NOGUEIRA, 2020)

Por fim, existem os nós de objeto que são utilizados para representar dados reais dentro dos digramas. Os principais são os nós de parâmetro da atividade, que representam valores de entrada ou saída das atividades e os de *pin*, que provêm valores de entrada para uma ação, ou aceitam seus valores de saída.

As arestas podem ser de dois tipos: de controle ou de objeto. A explicação para estas é simples. As arestas de controle ligam nós de controle, enquanto as de objeto ligam nós de objeto.

Um dos principais problemas da UML é sua semântica descrita de forma textual, o que abre espaço para ambiguidades, dificultando a verificação de modelos. Para resolver este problema, muitos dos trabalhos atuais buscam prover semânticas formais para a UML, fazendo com que os diagramas possam ser traduzidos e verificados utilizando linguagens e métodos formais, já que estes métodos são rigorosos quanto às suas semânticas e não-ambíguos. Neste trabalho utilizaremos a semântica fornecida no trabalho de (LIMA; TAVARES; NOGUEIRA, 2020).

### 2.1.2 Diagramas de Máquina de Estado

Quando o comportamento de uma entidade não depende somente de entradas ou saídas, mas também é necessário considerar seus estados passados, é possível modelá-lo com um diagrama de máquina de estado.

(PARADIGM, 2022) define estado como uma abstração dos valores de atributo e links de um objeto. Isto é, o estado corresponde à situação do objeto em um dado momento do tempo.

A especificação de (GROUP, 2017) nos diz que um diagrama de máquina de estados é composto por estados, que podem ser de diferentes tipos, decisões e transições. Explicamos abaixo cada um destes, com a visualização na Figura 7.

- Estado inicial: Define o começo de um comportamento de objeto.
- Estado final: Define o fim de um comportamento de objeto.
- Estado de decisão: Representa uma decisão a ser tomada com base no estado atual.
- Transição: Representa a mudança de um estado para outro. Elas podem ter três elementos: o gatilho, que é o motivo de a transição ocorrer; a guarda (*trigger*), que é uma condição necessariamente verdadeira para que o gatilho seja acionado e; por fim, a ação, que é invocada diretamente no objeto que possui a máquina de estado como resultado da transição.
- Estado: Representa as condições atuais de um objeto em determinado ponto do tempo. Pode ser do tipo simples, que não tem nenhuma subestrutura, do tipo composto, que contém outros estados dentro de si ou do tipo submáquina, que é similar ao anterior, mas pode ser reutilizado.

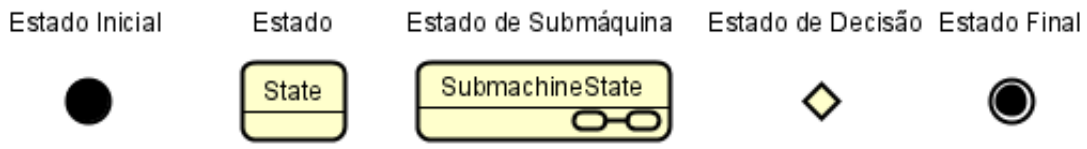


Figura 7 – Diferentes tipos de estados

## 2.2 Model-Based System Engineering

O modelo tradicional de engenharia de sistemas é baseado em documentos. Conforme os sistemas crescem, torna-se inviável manter tais documentos consistentes, completos e válidos. Isso acontece pois os documentos são criados e atualizados por múltiplos *stakeholders*, tornando sua manutenção cara e demorada (FERNANDEZ; CORBATO, 2019).

(FERNANDEZ; CORBATO, 2019) define A MBSE como “a aplicação formalizada de modelagem para apoiar os requisitos do sistema, projeto, análise, verificação e atividades de validação começando na fase de projeto conceitual e continuando ao longo do desenvolvimento e fases posteriores do ciclo de vida”. Ou seja, a *Model-Based*

*System Engineering* (MBSE) tem como objetivo corrigir os problemas da abordagem tradicional, utilizando-se de modelos em todas as fases de projeto.

De maneira geral, um modelo pode ser dado como uma representação de um elemento do mundo real. Na engenharia de sistemas, um modelo é uma abstração de um sistema e pode ser utilizado para comunicação e simulação. Este primeiro uso está relacionado com a transferência de informação entre as pessoas do projeto, podendo ser um diagrama, ou até mesmo representações em imagens de partes do sistema. Já o segundo é utilizado para simular o sistema e um dos exemplos deste uso é o diagrama de atividades, explicado na seção anterior.

Para a MBSE, é fundamental a utilização de ferramentas que auxiliam nas atividades do processo, como a construção e manutenção de modelos, além de sua verificação e validação. Dentro destas atividades, existe a Verificação de Modelos, a qual é um dos propósitos do nosso trabalho e explicamos de maneira mais detalhada na Subseção 2.2.1.

### 2.2.1 Verificação de Modelos (*Model Checking*)

A checagem (ou verificação) de modelos é uma técnica para automatizar a validação de propriedades sistemas. (CLARKE, 1997) nos mostra que comparada às técnicas utilizadas anteriormente, baseadas na prova automatizada de teorema,a verificação de modelos traz diversas vantagens. Basta prover uma representação de alto nível para determinada especificação, que os verificadores retornarão um resultado a respeito do atendimento ou não à especificação. Caso a especificação seja atendida, meramente se retorna um "*true*", caso contrário, retorna-se um contraexemplo.

Em sua fase inicial, (CLARKE, 1997) nos diz que os verificadores de modelo eram extremamente básicos, sendo capazes somente de validar pequenos sistemas. Atualmente diversos trabalhos desenvolveram o tema, como o de (VISSER; DWYER; WHALEN, 2012), que empenha técnicas de métodos formais na validação de modelos.

De acordo com (ROSCOE; DAVIES, 2011), a *Communicating Sequential Processes* (CSP) é uma notação matemática projetada para descrever interações. Esta notação pode ser usada para descrever comportamentos. Apesar de sua aparente utilidade, essa notação não pode ser lida por um computador, por conta disso, (ROSCOE, 2005) desenvolveu uma nova versão da CSP, que pode ser entendida por computadores. Sobre a versão criada por (ROSCOE, 2005), foi construída a ferramenta *Failures-Divergences Refinement* (FDR) (BROWNLEE, 2000), que traduz a notação para um formato no qual é possível executar uma checagem exaustiva de propriedades.

Apesar de existirem ferramentas como o FDR, que auxilia na verificação de modelos, muitas das ferramentas propostas são *stand-alone*, oferecendo pouca intero-

perabilidade entre sistemas. Uma tendência recente é utilizar arquiteturas orientadas a serviços para disponibilizar funcionalidades de sistemas desse tipo na web.

## 2.3 Microsserviços

Uma tendência recente é a migração de sistemas monolíticos para uma arquitetura focada em serviços (AUER et al., 2021). Essa mudança está diretamente ligada à necessidade de se ter mais robustez e reutilização de código, além da necessidade de sistemas mais interoperáveis.

A arquitetura em microsserviços é uma área dentro desta tendência. Segundo (DRAGONI et al., 2018), ela é construída sobre alguns princípios, explicados abaixo:

- Limitação de Contexto: Funcionalidades que são relacionadas devem ser combinadas em um único caso de uso, que deve ser implementado em um único serviço. Esta abordagem permite um grande alinhamento entre casos de uso e a estrutura do sistema, tornando fácil a manutenção.
- Tamanho: Existe a necessidade de manter os serviços pequenos, caso um deles seja muito grande é necessário refatorar em serviços menores. Os benefícios desta característica estão relacionados a manutenibilidade e extensibilidade.
- Independência: Esta característica dita que os serviços devem estar fracamente acoplados. Isto é, a operação de um serviço deve ser independente de outros serviços. Isso permite uma maior modificabilidade, tornando possível realizar mudanças no sistema mantendo-o correto.

Tendo em vista estes conceitos, a arquitetura de microsserviços consegue prover diversos benefícios relacionados a escalabilidade. Ainda de acordo com (DRAGONI et al., 2018), os principais são:

- Portabilidade: Como os microsserviços são distribuídos em *containers*, é possível replicá-los facilmente em diversos ambientes de múltiplas plataformas.
- Elasticidade: Dada a natureza “containerizada” dos microsserviços é muito simples escalá-los dinamicamente.
- Disponibilidade: A habilidade de replicá-los em múltiplos servidores em qualquer distância um do outro possibilita um balanceamento de carga não atingível em modelos monolíticos.
- Robustez: As características acima auxiliam bastante na tolerância de falhas, aumentando a robustez do sistema como um todo.

É comum utilizar-se uma API *Gateway* quando se está lidando com microsserviços. (GADGE; KOTWANI, 2018) define a API *Gateway* como um ponto de entrada único para o sistema. Além da vantagem de se ter um único *entry-point*, ela também auxilia na abordagem de alguns problemas, como o de *caching* e segurança, especialmente no nível de autenticação. É muito mais simples ter que autenticar um cliente em um único ponto do que em cada um dos serviços.

Neste contexto, (BIEBER, 2001) explica serviços como sendo comportamentos definidos através de um contrato. Estes comportamentos podem ser implementados e fornecidos por qualquer componente para uso por qualquer componente, com base exclusivamente no contrato.

Para se especificar estes contratos, é comum utilizar ferramentas como o Swagger (SOFTWARE, 2022), que gera a documentação de forma automática e interação com ela através de páginas web.

Uma estrutura típica de arquitetura de microsserviços é mostrada na Figura 8, na qual os clientes (que pode ser dispositivos móveis, computadores e até mesmo outros serviços), fazem uma chamada à API *Gateway*, que agrupa os serviços, que por sua vez fazem acesso a uma base de dados.

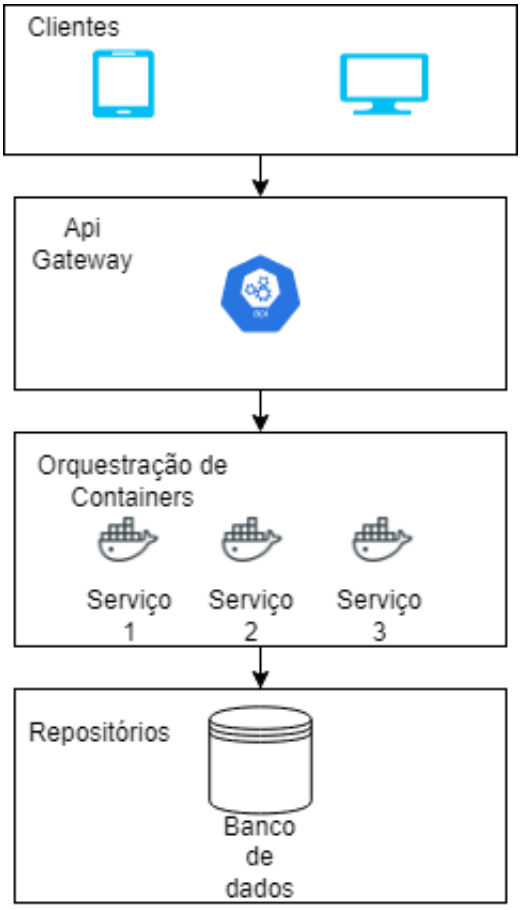


Figura 8 – Exemplo de uma arquitetura em microsserviços

# 3 Serviço de Verificação de Modelos Comportamentais UML

Nesta seção, iremos apresentar todos os serviços criados para a validação de diagramas UML que foram desenvolvidos. Na Seção 3.1, damos uma visão geral sobre a arquitetura dos serviços. A Seção 3.2 apresenta em mais detalhes como os serviços foram construídos.

## 3.1 Visão Geral da Arquitetura dos Serviços

Os serviços foram desenvolvidos com base no *framework* proposto em (LIMA; TAVARES; NOGUEIRA, 2020) e seguem a arquitetura mostrada na figura 9.

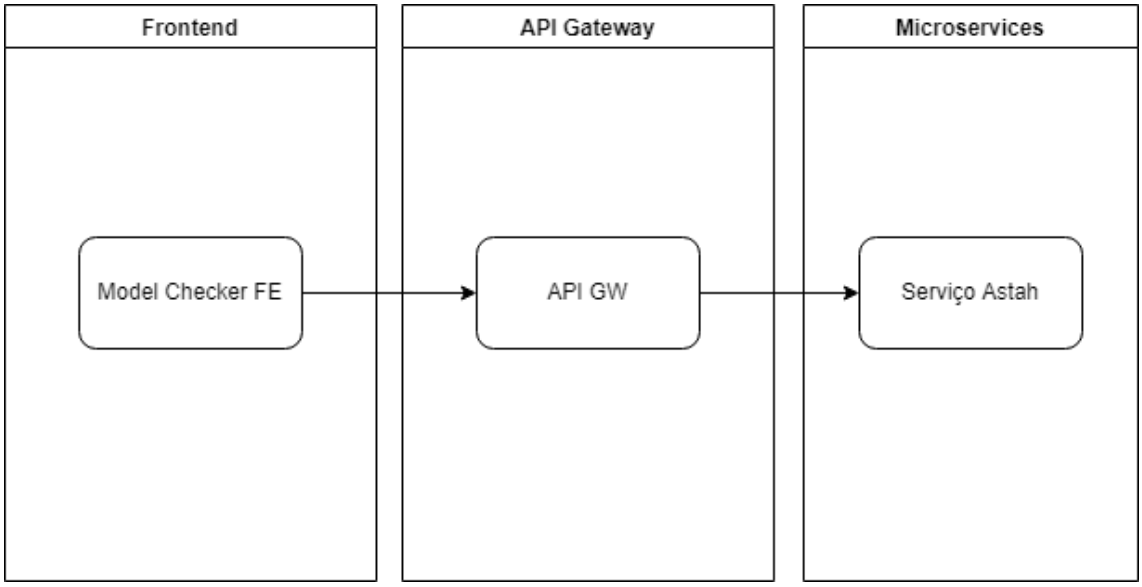


Figura 9 – Visão Geral do Serviço

Foi criada uma página web para acesso rápido aos serviços, que não é essencial, pois é possível consumir os serviços diretamente através da API *Gateway*, mas auxilia na utilização de usuários menos técnicos. Nesta página basta fazer *upload* do arquivo que se deseja validar, escolher o tipo de validação, selecionar o diagrama e submetê-lo. Esta etapa na Figura 9 corresponde ao “Model Checker FE” e está representada na Figura 10, na qual está sendo validado um arquivo chamado Aspirador, que contém um diagrama de mesmo nome. O tipo de validação selecionado foi o de determinismo.

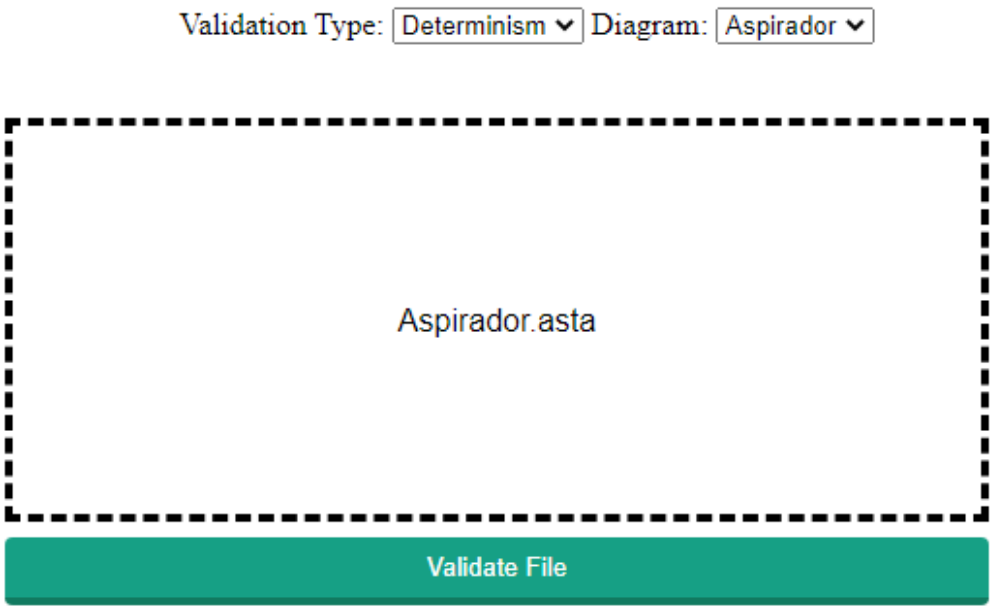


Figura 10 – Front-end criado para a validação de diagramas.

Após esta etapa, ao selecionar a validação, é feita uma requisição HTTP à API *Gateway*. Escolhemos essa abordagem por aumentar a extensibilidade do nosso serviços. Com ela é possível futuramente inserir novos serviços que cuidem da validação de outros tipos de diagrama, por exemplo, ou até mesmo em outros formatos.

Baseada na rota solicitada pelo cliente, a API *Gateway* encaminha para o microsserviço. Neste caso, o de validação de diagramas feitos na plataforma Astah. Os detalhes dos microsserviços serão apresentados na próxima seção, mas, de forma resumida, após realizar a validação do diagrama selecionado, é retornado para o cliente um arquivo com o contra-exemplo, caso exista.

A exemplificação desta utilização está presente na Figura 11, seguindo a ramificação de “Usuários em Browsers”.

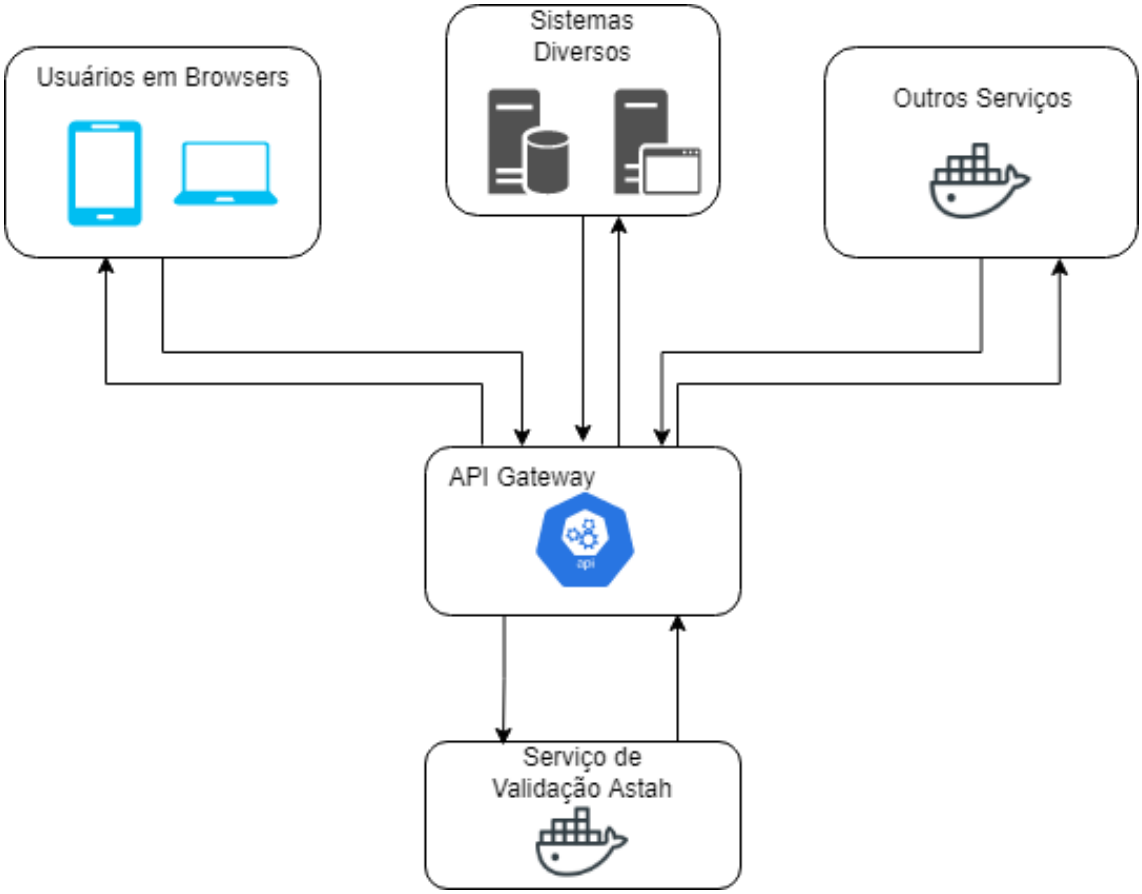


Figura 11 – Diversas formas de utilização do Serviço

### 3.1.1 Tecnologias Utilizadas na Implementação

Nesta Subseção descreveremos as tecnologias utilizadas na implementação do serviço. A tabela abaixo apresenta as tecnologias de acordo com cada camada.

| **Camada** | **Tecnologias** |
|---|---|
| Frontend | HTML, CSS e JavaScript |
| API Gateway | Spring Framework (Java) |
| Serviço de Validação | Spring Framework (Java) |

A camada de *frontend* foi criada para demonstrar uma forma rápida na qual um usuário qualquer poderia consultar o serviço de validação. Para esta camada utilizamos HTML, CSS e JavaScript, sem nenhum *framework*, pois a construção foi de somente uma página web que realiza poucas consultas. Para a atualização dinâmica da página após a realização de requisições HTTP utilizamos *Asynchronous JavaScript and XML* (AJAX) (GARRETT, 2005).

Para a construção da API *Gateway* e serviço, utilizamos o *framework* Spring (JOHNSON et al., 2004), que é uma solução leve para construção de aplicações web. A escolha do Spring está diretamente relacionada à necessidade de execução de código Java de maneira nativa.

Adicionalmente, para controle de versão foi utilizado o Git (CHACON, 2022), o código está hospedado no GitHub (INC, 2022) e disponível em (CAVALCANTI, 2022). Ainda no repositório em que está hospedado o código existem as instruções necessárias para permitir que os interessados possam instalar este serviço em suas infraestruturas.

## 3.2 Estrutura detalhada dos microsserviços

Como descrito anteriormente, utilizamos a arquitetura criada em (LIMA; TAVARES; NOGUEIRA, 2020) para realizar a validação dos diagramas. Dessa forma, o microsserviço segue a estrutura mostrada na Figura 12.

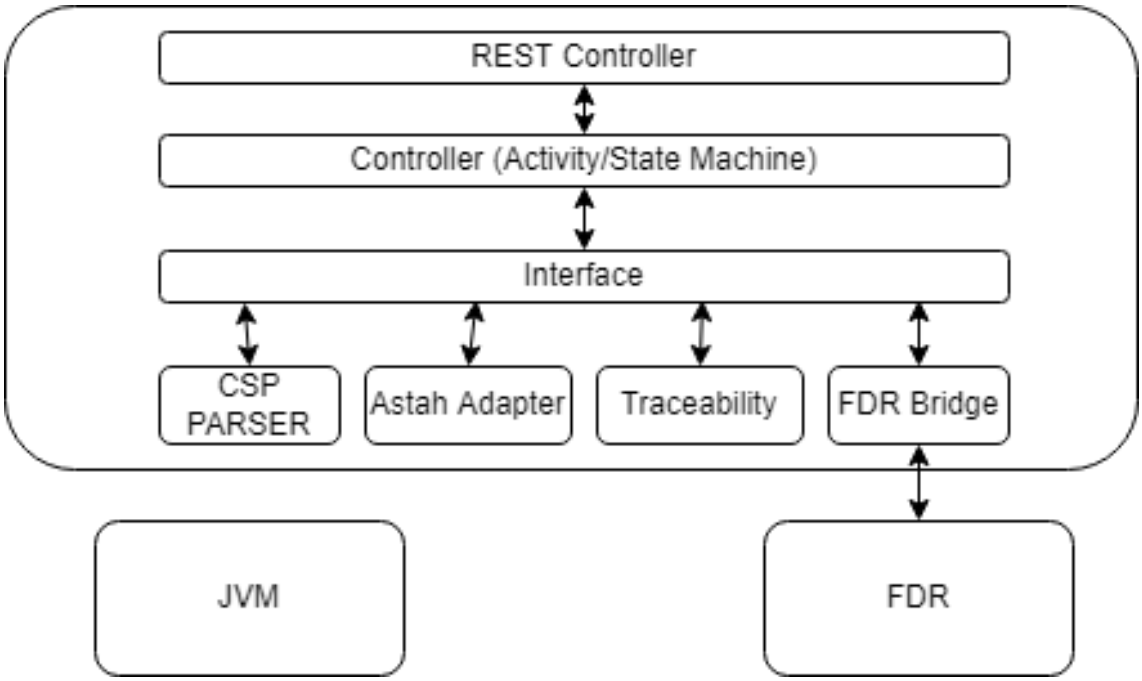


Figura 12 – Arquitetura do microsserviço em detalhes

A primeira camada da arquitetura, nomeada de “REST Controller” representa um controlador, que recebe as requisições HTTP e lida com elas. Este controlador tem diversos métodos construídos. Dentre eles, estão disponíveis as funcionalidades de descobrir quais diagramas estão presentes no arquivo e a de validação de um diagrama.

Para a descoberta de diagramas utilizamos a API disposta pela própria ferramenta Astah (VISION, 2022), que contém vários métodos utilitários para lidar com diagramas.

Já para a validação, nosso controlador comunica-se com outros dois controladores, os de Diagrama de Atividade e Máquina de Estados. Estes últimos são responsáveis por realizar a comunicação entre os módulos do *framework* e retornar os contraexemplos, quando existirem. Na figura, eles são representados pelo módulo “*Controller (Activity/State Machine*”.

O módulo de interface, por sua vez, tem como responsabilidade prover uma API que descreve os elementos dos diagramas. É neste módulo que estão descritos, por exemplo, os estados de uma máquina de estados e os nós de um diagrama de atividades.

No módulo *Adapter*, são feitas as implementações das interfaces descritas no módulo anterior. Com estas implementações é possível utilizar o módulo ”CSP Parser”, que faz a tradução dos objetos que representam atividades ou máquinas de estado para código CSP, seguindo suas especificações.

Além destes, o módulo de ”FDR Bridge”é responsável por realizar a comunicação com a ferramenta FDR. Esta ferramenta retorna os eventos que ocorreram durante sua utilização para nosso módulo.

Estes eventos são, por fim, utilizados no módulo de *traceability*, que cria os contraexemplos e os retorna para o controlador, quando necessário.

O último elemento da Figura 12 que ainda não foi explicado é a Java Virtual Machine (JVM). (GOSLING et al., 2000), a define como uma máquina virtual que permite computadores executem programas escritos na linguagem Java. Necessária em nosso trabalho, pois todos os módulos foram escritos nesta linguagem.

## 3.3 Contrato dos Serviços

Para definir o contrato dos serviços utilizamos o Swagger (SOFTWARE, 2022). Aos inicializá-los, fica disponível uma página web que especifica o que é necessário para utilizar cada serviço, em relação aos parâmetros de entrada esperados. Nesta página também são especificadas as possíveis saídas dos métodos e é possível invocar os serviços nela. A Figura 13 mostra os contratos especificados.

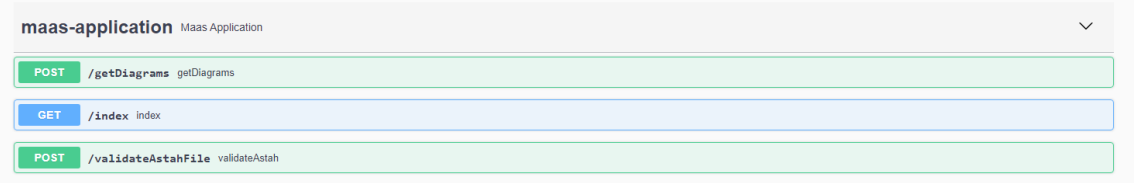


Figura 13 – Contratos do Serviço

Existem três métodos disponíveis: getDiagrams, index e validateAstahFile, que explicamos abaixo.

- getDiagrams: Este método foi criado para obter uma lista com todos os diagramas do arquivo astah submetido.
- index: retorna uma página HTML com o frontend da aplicação.

- validateAstahFile: é neste método em que fica a validação do diagrama. Ele espera como parâmetro o tipo de validação, o arquivo e o nome do diagrama a ser validado. O método se conecta com os controladores explicados na Seção 3.2, que devolvem o arquivo com o contraexemplo. Por fim, o método retorna o arquivo modificado contendo um contraexemplo, caso exista.

Para exemplificar a invocação dos métodos, mostramos abaixo a execução de um fluxo.

Ao acessar o endpoint /index, a página presente na Figura 14 é retornada.


Figura 14 – Retorno do endpoint /index

Nela, ao selecionar um arquivo astah, é feita uma requisição para ”/getDiagrams”, que por sua vez retorna a lista de diagramas presentes no arquivo, conforme a Figura 15.

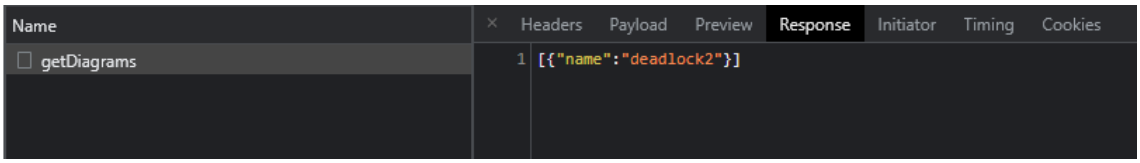


Figura 15 – Retorno do endpoint /getDiagrams

Por fim, ao selecionar o botão ”Validate File”, é feita uma requisição ao endpoint ”/validateAstahFile”, que retorna o arquivo validado, como mostrado na Figura 16.

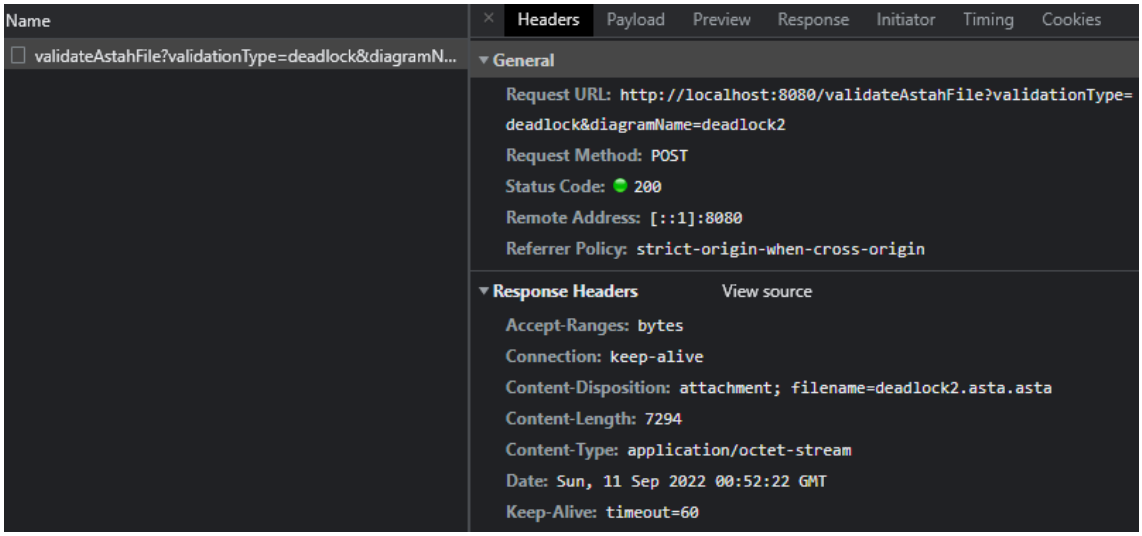


Figura 16 – Retorno do endpoint /validateAstahFile

Neste caso, o diagrama enviado foi o ”deadlock2”, que é o mesmo da Figura 5. O arquivo retornado está na Figura 17, que mostra uma aresta pintada de vermelho, onde é possível acontecer um deadlock.

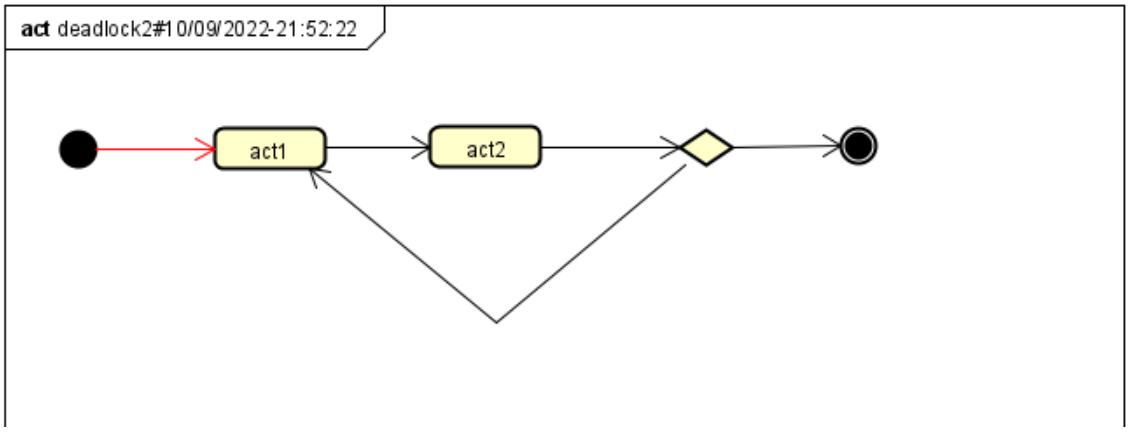


Figura 17 – Diagrama validado

# 4 Avaliação

Neste capítulo iremos apresentar a aplicação prática dos serviços de verificação de *deadlock* e não-determinismo em alguns estudos de caso. A Seção 4.1 irá tratar do uso com o *front-end* criado para o serviço, enquanto a Seção 4.2 apresentará o uso através da ferramenta de documentação Swagger. A seção 4.3 exemplifica o uso com a ferramenta de criação de APIs Postman e, por fim, é mostrada a utilização do serviço dentro de uma aplicação Windows, na Seção 4.4.

## 4.1 Utilizando Front-end Provido pelo Serviço

Nosso primeiro estudo de caso é o de um desenvolvedor que precisa utilizar de uma maneira rápida um serviço para validar seus diagramas. Neste caso, não é importante automatizar, somente ter o diagrama validado é o que importa.

Para atingir esse objetivo, ele decide utilizar a opção já fornecida pelo serviço que criamos. Sendo assim, ele acessa o *front-end* e submete seu arquivo Astah na caixa de *upload* (área tracejada), conforme mostrado na Figura 18, escolhe o arquivo que deseja validar e o diagrama em questão.

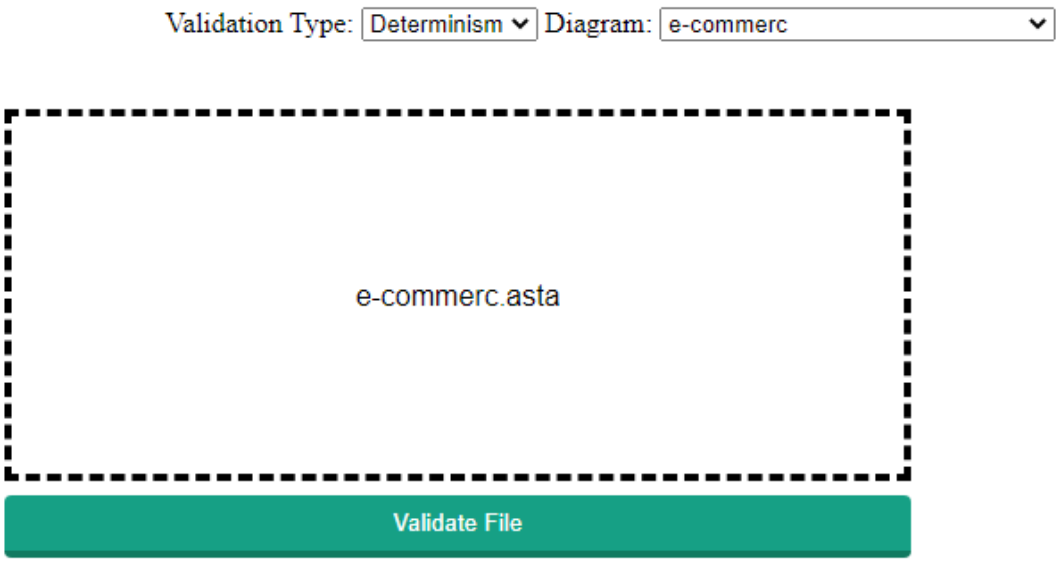


Figura 18 – Utilização do front-end padrão do serviço

Nesse estudo de caso, o diagrama selecionado é o de uma plataforma de comércio *online*, que foi apresentado no trabalho de (LIMA; TAVARES; NOGUEIRA, 2020). O diagrama que corresponde a este sistema pode ser visualizado na Figura 19.

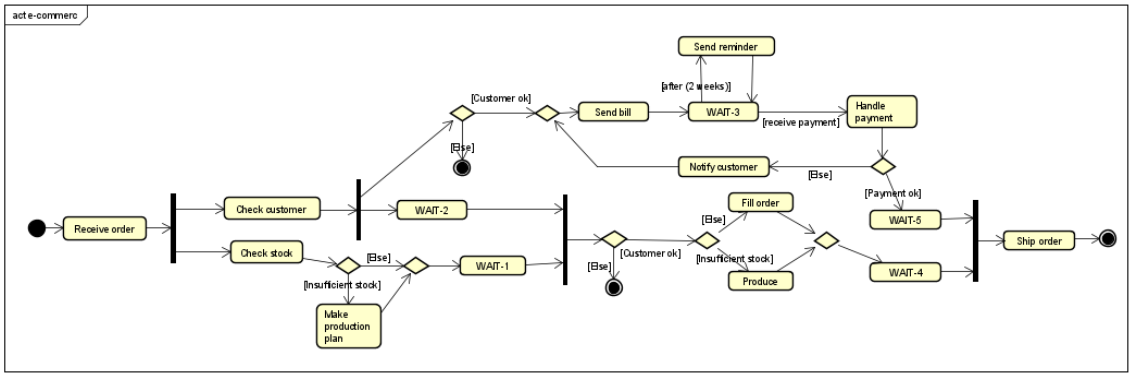


Figura 19 – Diagrama de um sistema de comércio online

A atividade começa no nó inicial (círculo preto preenchido), e passa para o nó seguinte "*Receive Order*", que corresponde ao recebimento do pedido de um cliente. Após o pedido ser recebido, acontece uma bifurcação no fluxo, que cria dois fluxos paralelos. O primeiro destes fluxos, que corresponde ao da parte de cima da Figura 19, é uma verificação do cliente, que se divide em outros dois fluxos que executam paralelamente. Destes fluxos de verificação do cliente, um checa se o cliente confirmou o pedido. Já o outro lida com o pagamento do pedido, enviando a conta no nó "*Send Bill*" e, caso o pagamento não tenha sido realizado dentro de duas semanas, enviando um lembrete. De volta aos dois fluxos iniciais, o segundo deles verifica a disponibilidade do produto no estoque, no nó "Check Stock". Caso não existam produtos disponíveis é criada uma ordem de produção, no nó "*Make Production Plan*". Após isso, o fluxo aguarda a finalização daqueles que estavam em execução paralela.

Após a realização do pagamento, acontece o tratamento deste no nó "*Handle payment*". Quando o pagamento não é aprovado, o cliente é notificado através do nó "*Notify Customer*", caso seja aprovado, o fluxo segue para aguardar a finalização do fluxo do produto, que executa em paralelo. Neste último, caso estejam disponíveis os produtos em estoque é realizado o preenchimento dos dados do pedido no nó "*Fill Order*". Após isso, o nó "*Ship Order*" é executado, no qual o pedido é enviado ao cliente, finalizando assim o fluxo da ação.

Quando executamos a verificação é encontrado um *deadlock* na nossa atividade, que pode ser visualizado na Figura 20. O caminho até o ponto problemático está marcado de vermelho, tornando fácil a identificação do problema, que é o nó "*WAIT-3*". Uma possível forma de resolver esse problema, seria adicionando um nó de *merge* antes do "*WAIT-3*". Dessa forma, ele não dependeria de duas entrada para executar, seria necessária somente uma.

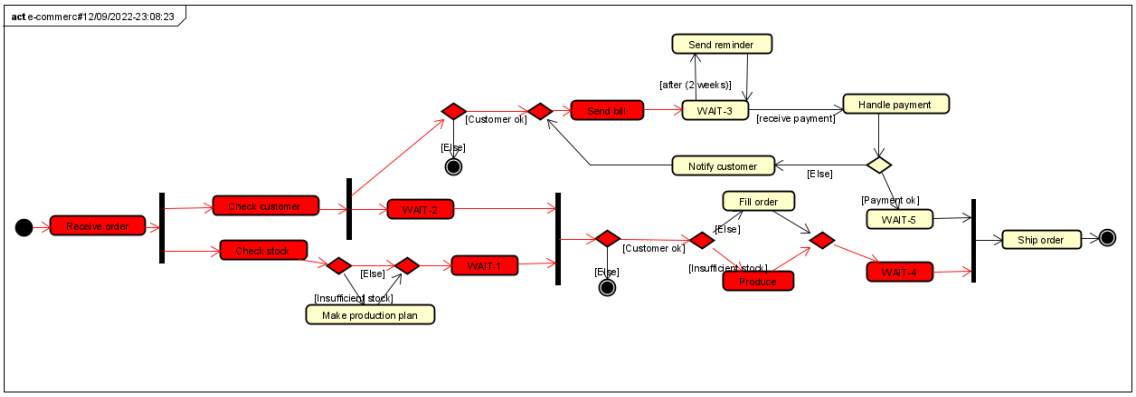


Figura 20 – Diagrama do sistema de e-commerce após verificação de deadlock

## 4.2 Utilizando Swagger

O segundo estudo de caso é o de um desenvolvedor que está buscando uma opção de serviço para validação de diagramas. Ele encontra o que construímos e acessa a documentação através do Swagger (SOFTWARE, 2022). Lá, tendo um arquivo Astah em mãos, ele resolve verificar se o serviço atenderia sua demanda. Ao acessar a documentação, ele preenche os dados, conforme a Figura 21. Os dados necessários para realizar a requisição são "*file*", que corresponde ao arquivo astah que contém o diagrama, após isso, "*diagramName*", que é o nome do diagrama a ser validado e, então, "*validationType*", que é o tipo de validação a ser executada.

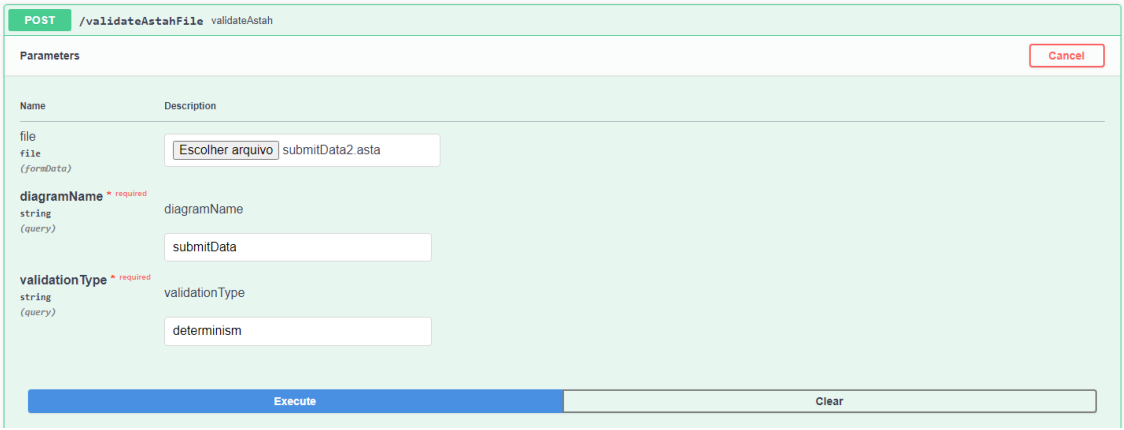


Figura 21 – Diagrama do sistema de e-commerce após verificação de deadlock

Este estudo de caso é o de um sistema de armazenamento de arquivos em base de dados e também foi retirado de (LIMA; TAVARES; NOGUEIRA, 2020). O diagrama correspondente a esse sistema está na Figura 22.

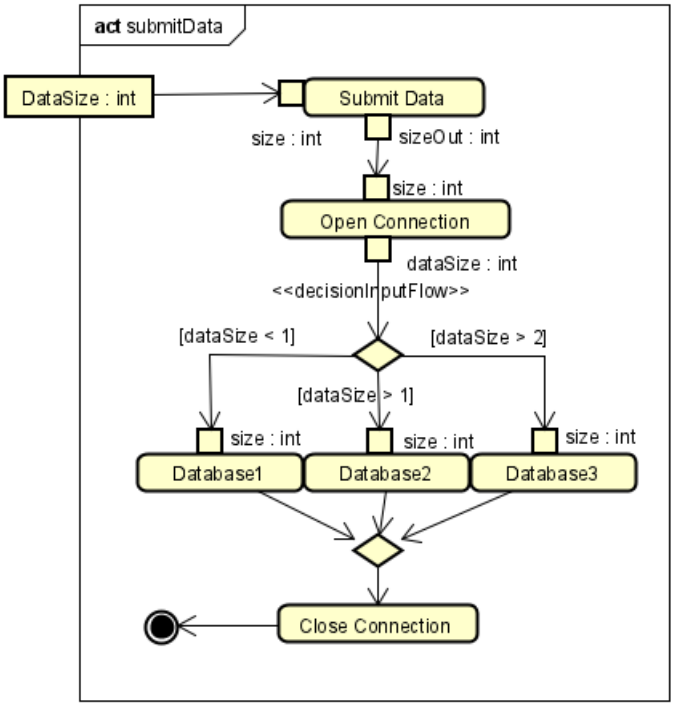


Figura 22 – Diagrama de um sistema de armazenamento de arquivos

O fluxo aqui começa no nó de paramêtro "*DataSize*", que corresponde a um número inteiro positivo maior do que 0. Ao receber uma entrada, o fluxo segue para o nó seguinte, "*Submit Data*", onde se inicia efetivamente a atividade. Após passar pelo nó anterior, o fluxo segue para o nó "*Open Connection*", responsável por abrir a conexão com os bancos de dados. Após a conexão ser estabelecida, o fluxo segue para um nó de decisão.

Este nó de decisão é responsável por definir para qual banco de dados o arquivo será enviado, baseado em seu tamanho. Aqui, a intenção é que arquivos de tamanho menor do que 1 sejam guardados no banco de dados 1 (nó *Database1*), arquivos de tamanho maior do que 1 fiquem no banco de dados 2 (nó *Database2*) e arquivos com tamanho superior a 2 fiquem no banco de dados 3 (nó *Database3*). Após o arquivo ser guardado, ele passa para o nó "*Close Connection*", que fecha a conexão com os bancos, encerrando a atividade após isso.

Quando selecionamos o botão "*Execute*", o Swagger (SOFTWARE, 2022), retorna o arquivo resultante, permitindo seu download, como na Figura 23.

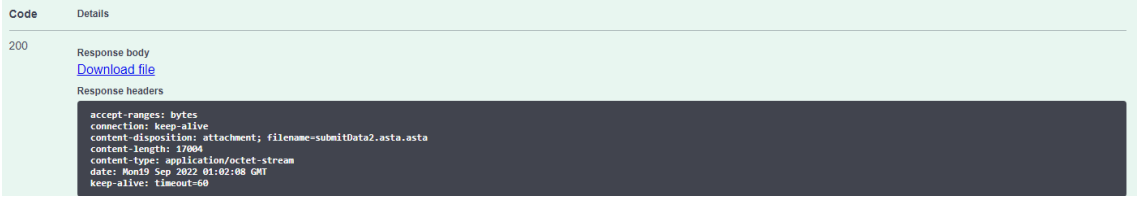


Figura 23 – Resultado obtido após executar a requisição via Swagger

Ao executarmos a verificação de não-determinismo, obtemos o diagrama atualizado com um contraexemplo, que pode ser visto na Figura 24, onde é possível observar a causa do não determinismo: as condições do nó de decisão. Neste nó, caso o *"dataSize"* seja maior que 2, ele pode ir tanto pelo caminho central quanto o caminho da direita de forma não-determinística.

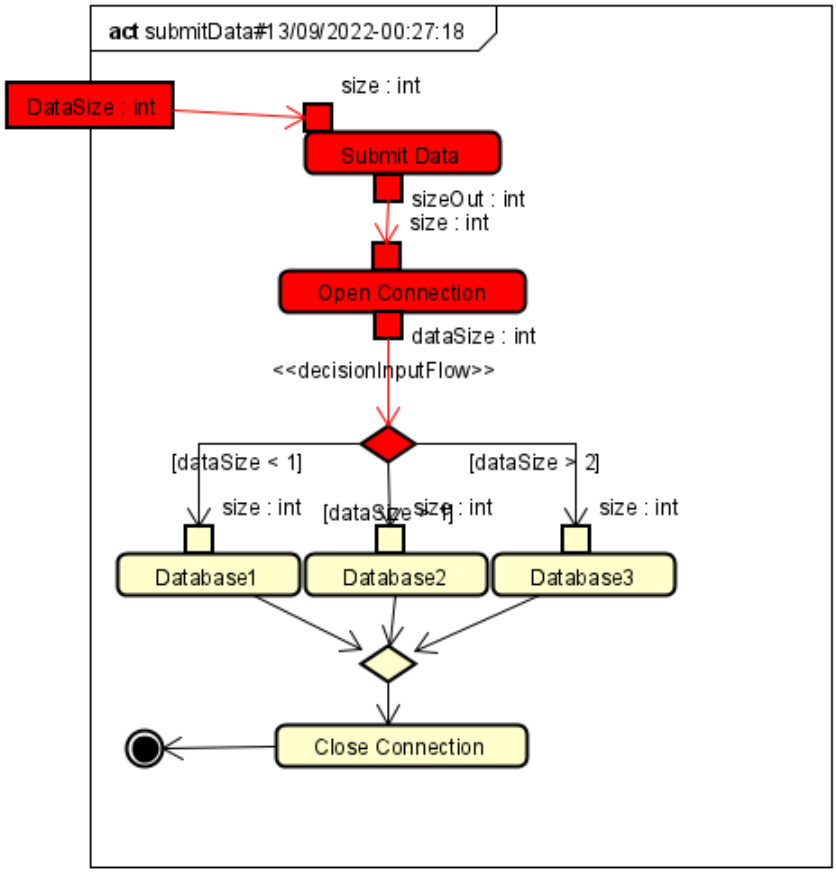


Figura 24 – Diagrama de um sistema de armazenamento de arquivos após verificação de não-determinismo

## 4.3 Utilizando Postman

No terceiro estudo de caso, um desenvolvedor já decidiu que irá utilizar os serviços que criamos e está validando seu uso através do Postman (POSTMAN, 2022), uma plataforma para construção e utilização de APIs.

Na interface do Postman, é criada a requisição, conforme mostrado na Figura 25. Nesta interface é selecionado o tipo da requisição, que é um "*POST*", depois disso é preenchida a URL completa para a qual será realizada a requisição. Pode-se notar que há dois parâmetros sendo passados nela, "*validationType*", que é o tipo de validação a ser executada, neste caso de determinismo e "*diagramName*", o nome do diagrama

no qual será executada a validação. Por fim, mais abaixo, existe um campo dentro do "*form-data*", que se chama *file*. Neste campo é passado o arquivo astah que contém o diagrama.

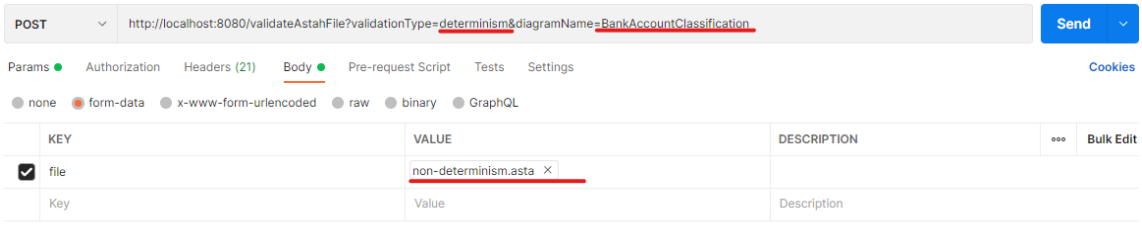


Figura 25 – Interface do Postman com requisição montada para validar um diagrama

Neste estudo de caso, temos um diagrama de máquina de estados, ele representa um sistema de classificação do tipo de uma conta bancária e foi criado por nós mesmos para este trabalho. O diagrama pode ser visto na Figura 26.

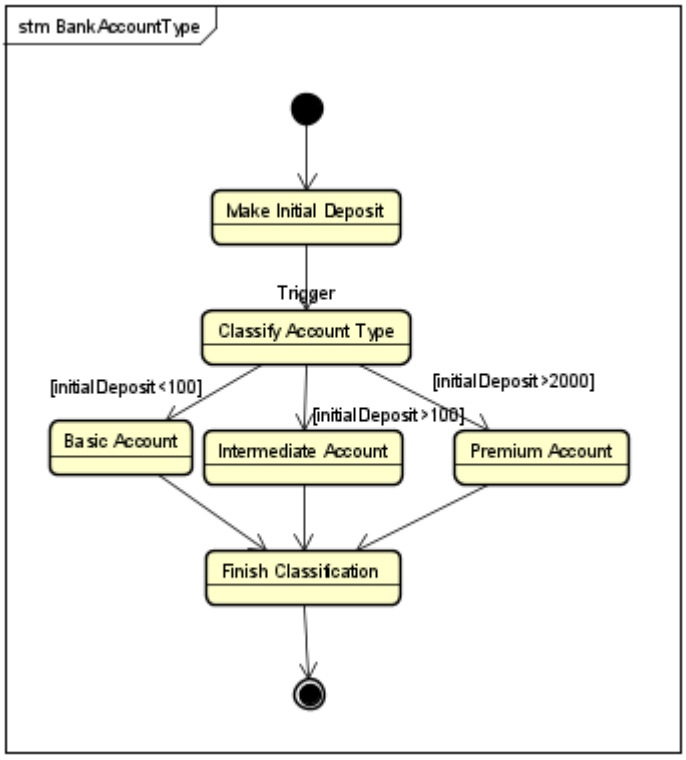


Figura 26 – Diagrama de um sistema de classificação de contas bancária

O fluxo se inicia ao fazer o depósito inicial na conta, no estado "*Make Initial Deposit*", após a realização deste depósito é acionado o gatilho para classificar o tipo de conta, no estado "*Classify Account Type*". Que, dependendo do valor depositado realiza uma transição para o tipo de conta básica ("*Basic Account*"), conta intermediária ("*Intermediate Account*") e conta premium ("*Premium Account*"). Após passar pelo

estado referente ao valor do depósito, é finalizada a classificação da conta e, então, a máquina de estado se encerra.

Ao enviar a requisição via Postman, o resultado é retornado em uma *string* de base 64, que pode ser salva como um arquivo, conforme podemos ver na Figura 27.

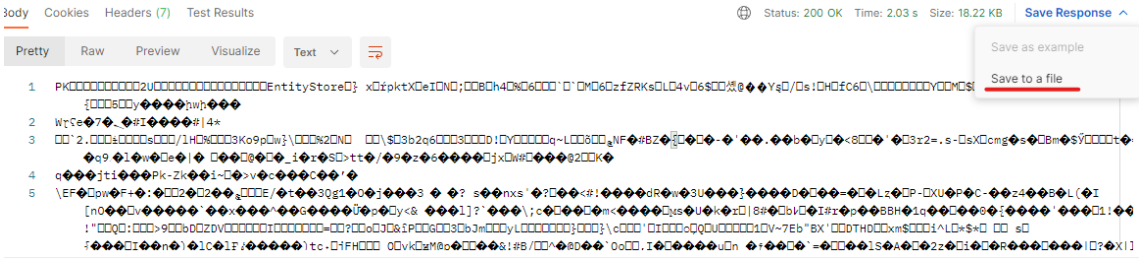


Figura 27 – Resposta da requisição feita pelo Postman

Quando executamos uma verificação em busca de não-determinismo neste diagrama, obtemos o resultado mostrado na Figura 28, na qual conseguimos perceber o problema deste diagrama: as guardas nas transições do estado “*Classify Account Type*” são inconsistentes, pois permitem realizar uma transição para “*Intermediate Account*” e “*Premium Account*” sempre que o valor do depósito inicial for maior que 2000. Um detalhe desta imagem é o estado marcado na cor roxa. Isso acontece pois o funcionamento do *trace* no validador na máquina de estados é diferente do diagrama de atividades.

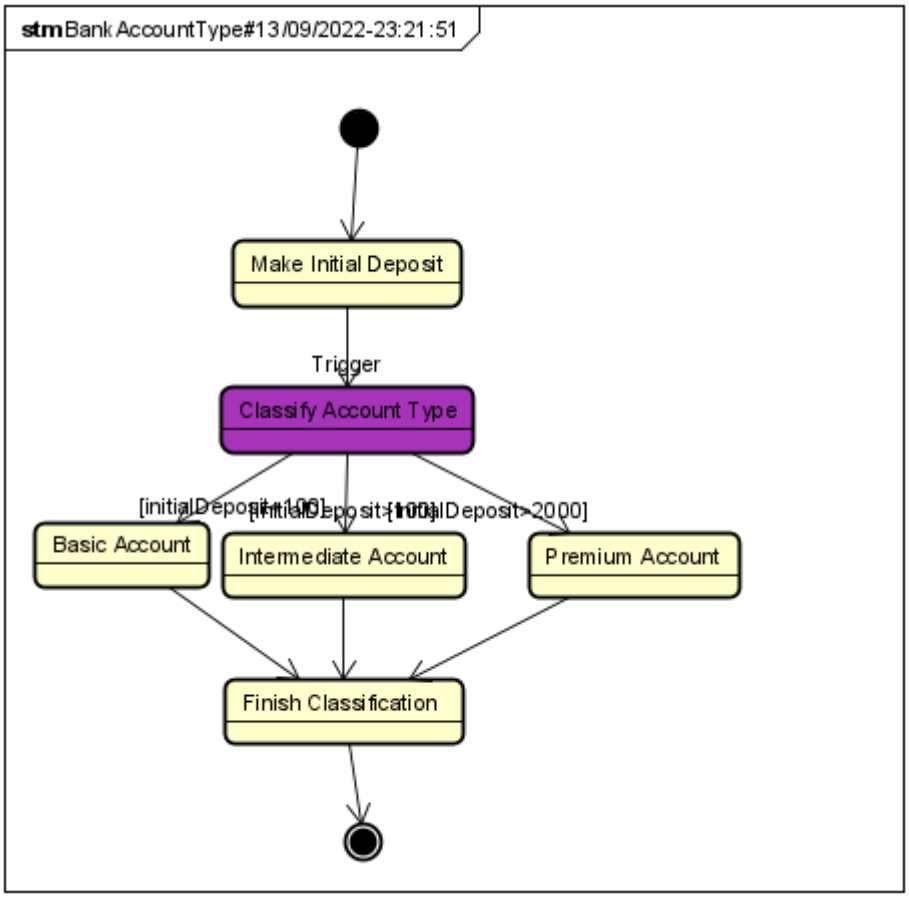


Figura 28 – Diagrama de um sistema de classificação de contas bancária após validação de não-determinismo

## 4.4 Utilizando Aplicação Desktop

Nosso último estudo de caso é o de um aplicativo *desktop*. Aqui, a intenção é que se tenha acesso rápido ao serviço sem a necessidade de acessar algum browser. Este caso também serve para exemplificar que é possível criar uma integração com outros sistemas já existentes para desktop. Aqui foi criada uma aplicação para *Windows* utilizando Electron (FOUNDATION, 2022), uma ferramenta que permite a utilização de HTML, CSS e JavaScript para construção de aplicativos *desktop*. Por ter sido escrito utilizando estas tecnologias, a aparência da aplicação é similar ao *front-end* desenvolvido para os serviços. A aplicação pode ser vista na Figura 29.

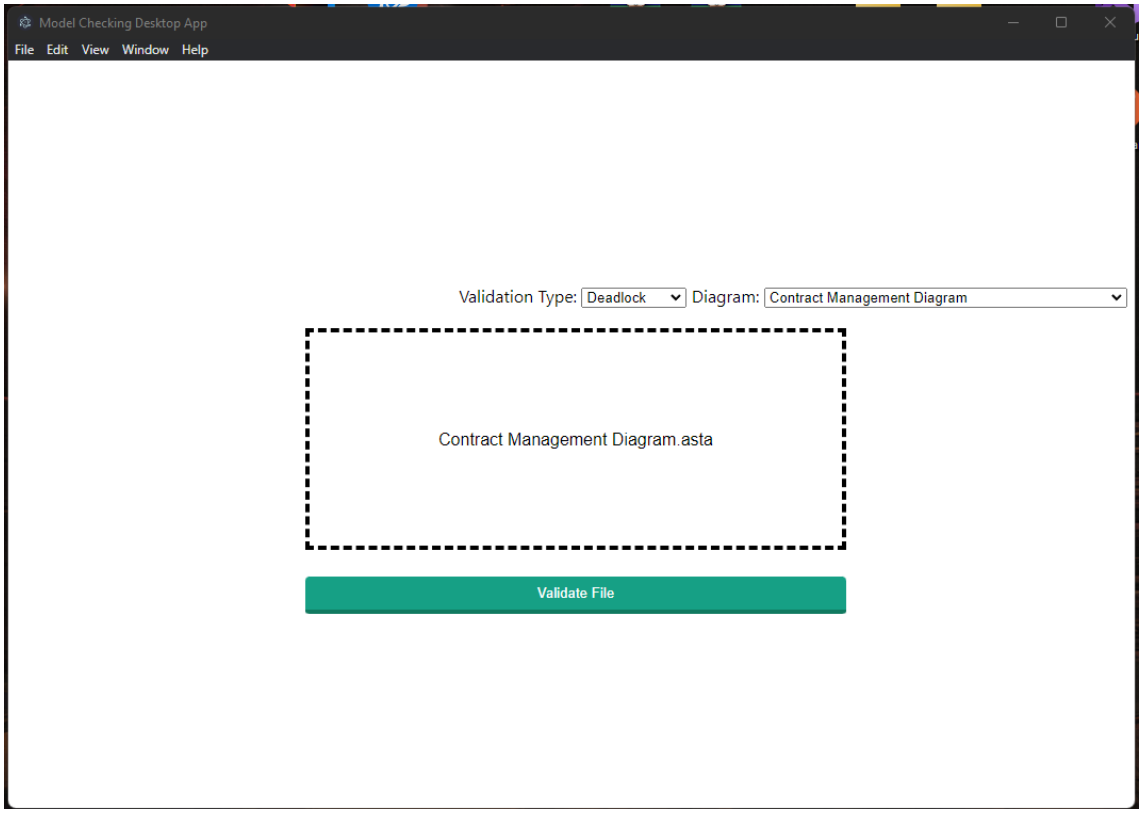


Figura 29 – Aplicação *desktop* para validação de diagramas

Este último estudo de caso é o de um sistema de gerenciamento de contratos. Neste sistema, os processos precisam passar por algumas etapas de aprovação até que o contrato seja finalmente aprovado. O diagrama correspondente a ele está presente na Figura 30.

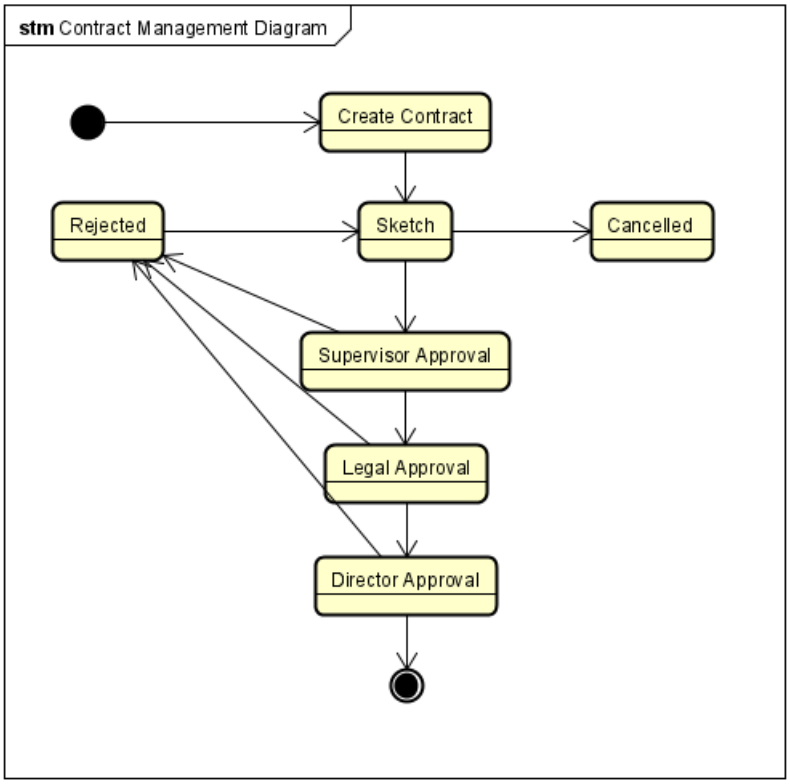


Figura 30 – Diagrama de um sistema de gerenciamento de contratos

O fluxo aqui se inicia quando algum funcionário cria um novo contrato. Após a criação, ele passa para o estado de rascunho ("*Sketch*") e a partir daí pode ser cancelado ("*Cancelled*") ou submetido para aprovação do supervisor ("*Supervisor Approval*"), que pode rejeitá-lo ou enviá-lo para a aprovação do jurídico ("*Legal Approval*"), que também pode rejeitar o pedido ou submetê-lo para a última etapa, de aprovação da diretoria ("*Director Approval*"). A partir daí, o contrato pode ser rejeitado ou finalizado. Quando no estado de rejeição ("*Rejected*"), o contrato ainda pode ser transformado em rascunho novamente, para que sejam feitas alterações necessárias para a aprovação.

Quando selecionamos o botão para realizar a checagem do diagrama, é possível salvar o arquivo validado em alguma pasta do sistema, como é mostrado na Figura 31

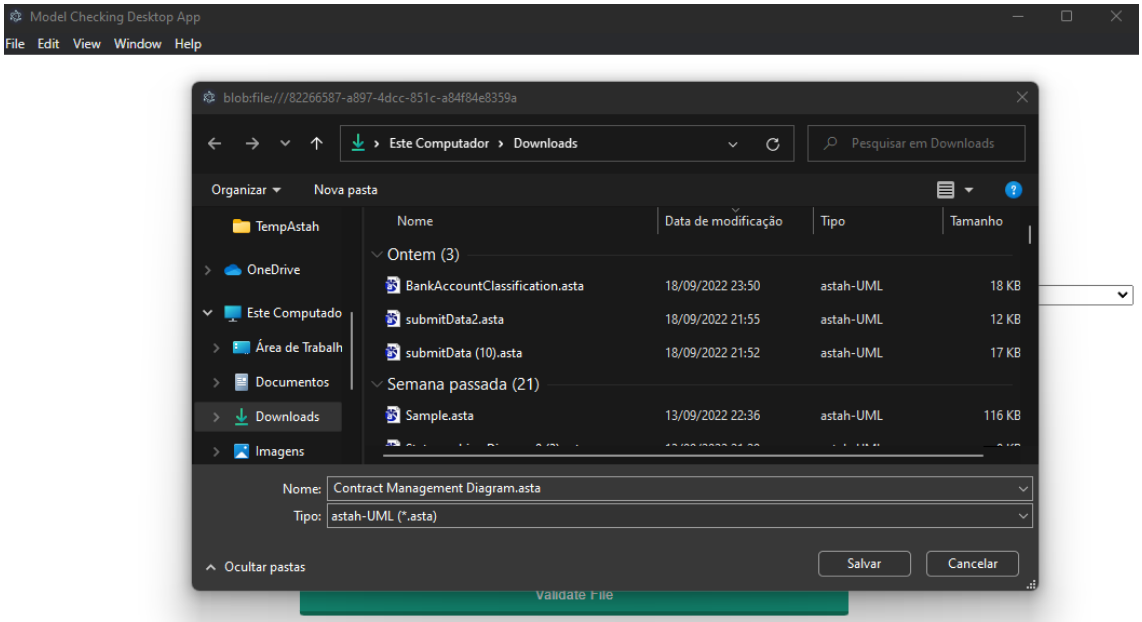


Figura 31 – Opção de salvar o arquivo validado dentro do sistema

Ao utilizar nosso serviço de validação de *deadlocks* neste diagrama, o arquivo retornado é o da Figura 32. Nessa figura é possível notar o problema do diagrama: ao passar para o estado de cancelado não existe nenhuma outra transição, nem para um estado final, fazendo com que a máquina fique em *deadlock*.

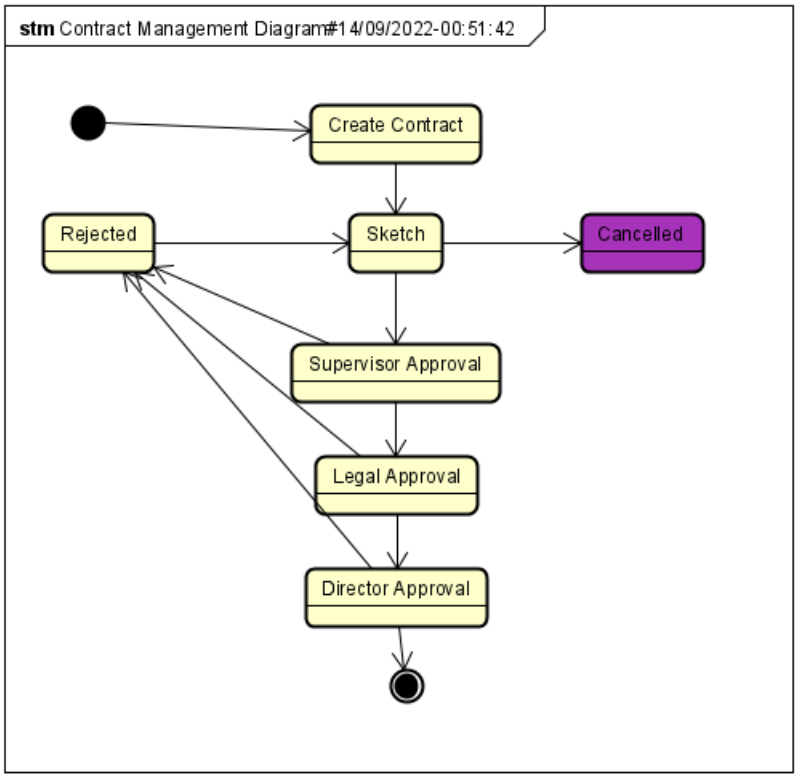


Figura 32 – Diagrama de um sistema de gerenciamento de contratos após verificação de deadlock

# 5 Conclusão

Neste trabalho nós construímos um serviço que permite a validação de diagramas UML na plataforma Astah. Nosso serviço verificador de propriedades utiliza métodos formais em suas camadas mais inferiores e permite a checagem de *deadlock* e não-determinismo em diagramas de Atividade e Máquina de Estado. A utilização de métodos formais envolve a conversão do modelo para linguagem CSP e, por fim, a utilização do FDR para checagem.

Apesar da complexidade de uma linguagem formal para descrição de comportamentos, neste trabalho conseguimos mantê-la distante do usuário final. Ou seja, nossa principal contribuição é permitir que modelos comportamentais tenham importantes características verificadas através de métodos robustos, sem a necessidade do conhecimento deles por parte do usuário.

Outro ponto importante é o de que o serviço disponibilizado é de maneira gratuita e de código aberto. Sendo assim, é possível consumi-lo facilmente ao ler sua documentação e também se pode fazer contribuições, para torná-lo mais completo. Tendo isto em conta, recomendamos a utilização do serviço sempre que for necessária uma verificação de *deadlock* ou não-determinismo em diagramas de atividade ou máquina de estado, no momento, só os que são construídos na plataforma Astah.

## 5.1 Trabalhos Relacionados

As abordagens apresentadas nesta seção também tem como objetivo realizar a verificação de modelos, ocultando do usuário os métodos formais que são utilizados. Porém, em sua maioria, são dependentes de uma plataforma específica.

O trabalho de (LIMA; TAVARES; NOGUEIRA, 2020) propõe um *framework* para verificação de *deadlock* e não-determinismo. Nosso trabalho se baseia fortemente neste, porém ele está limitado à ferramenta Astah (VISION, 2022) e somente permite verificação de diagramas de atividade.

A abordagem de (HORVÁTH et al., 2020) também propõe um serviço de verificação de modelos. Esta abordagem também utiliza uma linguagem intermediária, a *Gamma Statechart Language* (GSL), que por fim passa por um verificador. O trabalho de (HORVÁTH et al., 2020), consegue verificar somente a propriedade de alcançabilidade, utilizando a linguagem SysML.

O trabalho de (OUCHANI; MOHAMED; DEBBABI, 2014) utiliza uma abordagem de tradução para uma linguagem intermediária, neste caso a PRISM. Essa abordagem

permite que seja verificada a propriedade de *deadlock*, mas não é possível verificar não-determinismo. Além disso, ela é dependente da instalação da ferramenta MagicDraw, que necessita de licença.

(BANTI; PUGLIESE; TIEZZI, 2011) propõem em seu trabalho uma ferramenta chamada *Verification ENvironment for UML models of Services* (VENUS), que permite a verificação de propriedades em diagramas de atividade da UML 2.0. A ferramenta alcança seu objetivo de utilizar métodos formais para a verificação sem requerer conhecimento do usuário, porém, é incapaz de retornar os pontos do diagrama passíveis de erros, além de ser necessária a instalação da ferramenta.

Em seu trabalho, (OBER; GRAF; OBER, 2004) apresentam uma forma de transformar diagramas UML em uma estrutura de comunicação de autômatos cronometrados estendidos. Após essa transformação, é utilizado um conjunto de ferramentas para verificação de propriedades. Apesar de conseguir esconder do usuário final os métodos formais, essa abordagem ainda requer do usuário o conhecimento de diversas ferramentas para que sejam executadas as validações.

## 5.2 Limitações e Trabalhos Futuros

Atualmente, o serviço criado somente fornece a possibilidade de verificação de *deadlock* e não-determinismo. A ferramenta utilizada, FDR, suporta outras verificações como refinamentos entre processos CSP. Isto permite também a verificação de propriedades mais genéricas, mas não abordamos isto neste trabalho.

Uma outra limitação que existe atualmente são as formas de interação com os serviços. Neste trabalho, só é possível ver os resultados da verificação tendo a aplicação do Astah instalada. Uma abordagem que permitisse obter a imagem do diagrama validado seria de grande valor.

Um outro ponto de melhoria é o de suporte a outras ferramentas. No estado atual, nosso serviço verificador somente consegue checar propriedades de diagramas UML construídos na plataforma Astah. A adição de suporte a outras ferramentas populares, como MagicDraw e Visual Paradigm seria uma melhoria importante, como também a ambientes de modelagem como o OpenMBEE (OPENMBEE, 2022).

Os tipos de diagramas suportados atualmente são os de atividades e máquina de estados. Porém, estes não são os únicos diagramas comportamentais existentes na UML. A possibilidade de verificação em outros tipos de diagramas comportamentais é um ponto de melhoria que pode ser explorado em trabalhos futuros.

Além destes pontos, um trabalho futuro poderia disponibilizar um contêiner docker com o serviço, permitindo que ele fosse facilmente instalado e disponibilizado.

# Referências


ANTHONY, S. I.; MARSLAND, T. A. The deadlock problem: An overview. *IEEE Computer*, v. 13, p. 58–78, 1980. Citado na página 15.

AUER, F. et al. From monolithic systems to microservices: An assessment framework. *Information and Software Technology*, v. 137, p. 106600, 2021. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584921000793>. Citado 2 vezes nas páginas 16 e 25.

BANTI, F.; PUGLIESE, R.; TIEZZI, F. An accessible verification environment for uml models of services. *Journal of Symbolic Computation*, v. 46, n. 2, p. 119–149, 2011. ISSN 0747-7171. Automated Specification and Verification of Web Systems. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0747717110001355>. Citado na página 47.

BARESI, L. Activity diagrams. In: ____. *Encyclopedia of Database Systems*. Boston, MA: Springer US, 2009. p. 41–45. ISBN 978-0-387-39940-9. Disponível em: <https://doi.org/10.1007/978-0-387-39940-9_9>. Citado na página 13.

BIEBER, G. Introduction to service-oriented programming ( rev 2 . 1 ) by guy bieber , lead architect , motorola isd jeff carpenter , software engineer , motorola isd. In: . [S.l.: s.n.], 2001. Citado na página 26.

BROWNLEE, N. Fdr2 user manual. In: *Blount MetraTech Corp. Accounting Attributes and Record Formats http://www.ietf.org/rfc/rfc2924.txt*. [S.l.: s.n.], 2000. Citado na página 24.

BUSHONG, V. et al. On microservice analysis and architecture evolution: A systematic mapping study. *Applied Sciences*, v. 11, p. 7856, 08 2021. Citado na página 16.

CAVALCANTI, P. *MaaS*. 2022. Disponível em: <https://github.com/Phnc/maas>. Citado na página 30.

CHACON, S. *Git –fast-version-control*. 2022. Git Website. Disponível em: <https://git-scm.com/site>. Citado na página 30.

CLARKE, E. M. Model checking. In: RAMESH, S.; SIVAKUMAR, G. (Ed.). *Foundations of Software Technology and Theoretical Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997. p. 54–56. Citado na página 24.

CLARKE, E. M. et al. (Ed.). *Handbook of Model Checking*. Springer, 2018. ISBN 978-3-319-10574-1. Disponível em: <https://doi.org/10.1007/978-3-319-10575-8>. Citado na página 12.

CLARKE, E. M.; WING, J. M. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 28, n. 4, p. 626–643, dec 1996. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/242223.242257>. Citado na página 13.

DRAGONI, N. et al. Microservices: How to make your application scale. In: PETRENKO, A. K.; VORONKOV, A. (Ed.). *Perspectives of System Informatics*. Cham: Springer International Publishing, 2018. p. 95–104. Citado na página 25.

FERNANDEZ, J.; CORBATO, C. H. *Practical Model-Based Systems Engineering*. [S.l.: s.n.], 2019. ISBN ISBN-13: 978-1-63081-579-0. Citado na página 23.

FOUNDATION, O. *Electron*. 2022. Electron Website. Disponível em: <https://www.electronjs.org/>. Citado na página 41.

FOWLER, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3. ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321193687. Citado na página 19.

GADGE, S.; KOTWANI, V. Microservice architecture: Api gateway considerations. *GlobalLogic Organisations, Aug-2017*, 2018. Citado na página 26.

GARRETT, J. J. Ajax: A new approach to web applications. In: . [S.l.: s.n.], 2005. Citado na página 29.

GOSLING, J. et al. *Java Language Specification, Second Edition: The Java Series*. 2nd. ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN 0201310082. Citado na página 31.

GROUP, O. M. *Unified Modeling Language, v2.5.1*. 2017. OMG Website. Disponível em: <https://www.omg.org/spec/UML/>. Citado 3 vezes nas páginas 20, 21 e 23.

HASKINS, B. et al. 8.4.2 error cost escalation through the project life cycle. *INCOSE International Symposium*, v. 14, p. 1723–1737, 06 2004. Citado 2 vezes nas páginas 12 e 13.

HORVÁTH, B. et al. Model checking as a service: Towards pragmatic hidden formal methods. In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. [S.l.: s.n.], 2020. Citado 3 vezes nas páginas 12, 15 e 46.

INC, G. *GitHub*. 2022. GitHub Website. Disponível em: <https://www.github.com>. Citado na página 30.

JOHNSON, R. et al. The spring framework–reference documentation. *interface*, v. 21, p. 27, 2004. Citado na página 29.

KOC, H. et al. Uml diagrams in software engineering research: A systematic literature review. *Proceedings*, v. 74, p. 13, 03 2021. Citado 2 vezes nas páginas 13 e 21.

LEWIS, J.; FOWLER, M. *Microservices a definition of this new architectural term*. 2014. Martin Fowler Blog. Citado na página 16.

LIMA, L.; TAVARES, A.; NOGUEIRA, S. C. A framework for verifying deadlock and nondeterminism in uml activity diagrams based on csp. *Science of Computer Programming*, v. 197, p. 102497, 2020. ISSN 0167-6423. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167642320301064>. Citado 8 vezes nas páginas 15, 16, 22, 27, 30, 34, 36 e 46.

LIONS, J.-L. Ariane 5 flight 501 failure: Report by the enquiry board. In: . [S.l.: s.n.], 1996. Citado 2 vezes nas páginas 12 e 16.

OBER, I.; GRAF, S.; OBER, I. Validation of uml models via a mapping to communicating extended timed automata. In: GRAF, S.; MOUNIER, L. (Ed.). *Model Checking Software*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 127–145. ISBN 978-3-540-24732-6. Citado na página 47.

OPENMBEE. *OpenMBEE - Open Model Based Engineering Environment*. 2022. OpenMBEE Website. Disponível em: <https://www.openmbee.org/>. Citado na página 47.

OUCHANI, S.; MOHAMED, O. A.; DEBBABI, M. A formal verification framework for sysml activity diagrams. *Expert Systems with Applications*, v. 41, n. 6, p. 2713–2728, 2014. ISSN 0957-4174. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0957417413008968>. Citado 2 vezes nas páginas 15 e 46.

PARADIGM, V. *What is Activity Diagram?* 2022. Visal Paradigm Website. Disponível em: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-activity-diagram/>. Citado 2 vezes nas páginas 21 e 22.

POSTMAN. *What is Postman?* 2022. Postman Website. Disponível em: <https://www.postman.com/>. Citado na página 38.

ROSCOE, A. *The Theory and Practice of Concurrency*. [S.l.: s.n.], 2005. Citado 2 vezes nas páginas 15 e 24.

ROSCOE, A. W.; DAVIES, J. Csp (communicating sequential processes). In: ____. *Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011. p. 478–482. ISBN 978-0-387-09766-4. Disponível em: <https://doi.org/10.1007/978-0-387-09766-4_186>. Citado na página 24.

SHEVCHENKO, N. *An Introduction to Model-Based Systems Engineering (MBSE)*. 2020. Carnegie Mellon University's Software Engineering Institute Blog. Citado na página 12.

SOFTWARE, S. *What is Swagger?* 2022. Swagger Website. Disponível em: <https://swagger.io/docs/specification/2-0/what-is-swagger/>. Citado 4 vezes nas páginas 26, 31, 36 e 37.

VISION, C. *Astah - Premier Diagramming, Modeling Software and Tools*. 2022. Astah Website. Disponível em: <https://astah.net/>. Citado 4 vezes nas páginas 15, 16, 30 e 46.

VISSER, W.; DWYER, M.; WHALEN, M. The hidden models of model checking. *Software & Systems Modeling*, v. 11, 10 2012. Citado 2 vezes nas páginas 13 e 24.