Universidade da Beira Interior
Departamento de Informática

Projeto 2023: *Modelling, Validation and Verification with Uppaal*

Elaborado por:

Eduardo Gonçalves

Orientador:

Professor Doutor Paul Crocker

17 de julho de 2023

Agradecimentos

Gostaria de expressar os meus sinceros agradecimentos a todos aqueles que tornaram possível a realização deste projeto. Em particular, quero agradecer à minha família pelo constante apoio e encorajamento, à universidade pela oportunidade de estudar e desenvolver as minhas habilidades académicas e profissionais e ao professor Paul Crocker pela orientação valiosa e inspiração ao longo deste projeto.

Sem o apoio dessas pessoas e instituições, este projeto não teria sido possível. Estou profundamente grato por todo o suporte que recebi e por ter tido a oportunidade de trabalhar em um projeto tão gratificante.

Conteúdo

Lista de Figuras

Lista de Excertos de Código

Acrónimos

UBI	Universidade da Beira Interior
TCTL	*Timed Computation Tree Logic*
URL	*Uniformed Resource Locator*
ICSEA	*International Conference on Software Engineering Advances*
SO	Sistemas Operativos
JDK	*Java Development Kit*

Glossário

Correctness Refere-se à precisão e validade de um programa de *software* ou componente em relação aos seus requisitos e especificações. 3, 4, 6, 9, 10, 60

Deadlock Ocorre quando dois ou mais processos ficam bloqueados, aguardando um pelo outro para libertar recursos que estão a ser utilizados. Isso resulta em uma paralisação do sistema, onde nenhum dos processos pode continuar sua execução. iv, v, vii, viii, 1, 9, 15, 24, 25, 28, 33, 38, 47–49, 51–53, 55, 57

Edges Em *UPPAAL*, representam as transições entre os estados do sistema, especificando as condições e ações necessárias para ocorrer a transição. 16, 22, 27, 31, 32, 36, 37, 39, 45, 49, 53, 55

Guards Em *Uppaal* guards são condições que são avaliadas em estruturas de controle, como *if-else* e *switch*, para decidir o fluxo de execução do programa. 16, 22, 27, 31, 32, 36, 39, 45, 46, 49–51, 53, 54

JAVA É uma linguagem de programação orientada a objetos, desenvolvida pela *Sun Microsystems* na década de 90. Hoje pertence à empresa *Oracle*. 7, 17

Mutual Exclusion É uma propriedade desejada em sistemas concorrentes, onde apenas um processo pode aceder um recurso compartilhado por vez, evitando assim condições de corrida e inconsistências nos dados. iv, v, 1, 15, 24–31, 33, 37–43, 47

OCaml É uma linguagem de programação funcional e imperativa, com suporte a tipagem estática. É amplamente utilizada no desenvolvimento de software de sistemas, linguagens de programação, análise de código e verificação formal. 7

Reachability É uma propriedade que indica se um estado específico pode ser alcançado a partir de um estado inicial em um sistema. A análise de alcançabilidade é frequentemente usada para verificar se um determinado estado é atingível ou inatingível. iv, v, 23, 29, 34, 38, 47

Safety É uma propriedade que garante que certas condições sempre sejam verdadeiras durante a execução de um sistema ou programa, independentemente das entradas ou eventos. Está relacionada à prevenção de falhas críticas ou violações de propriedades importantes. iv, v, 23, 28, 33, 47

***Updates*/Atualizações** Em *Uppaal updates* ou atualizações referem-se a modificações ou melhorias feitas em um programa ou sistema existente, geralmente para corrigir bugs, adicionar recursos ou aprimorar o desempenho. 22, 27, 31, 32, 36, 39, 40, 45, 46, 49–51, 53, 54

Utility É uma propriedade que prova que o sistema tem utilidade para comprovar as definições propostas. iv, v, 23, 28, 34, 38, 47

Análise de Fluxo de Dados É uma técnica de análise estática que estuda como os dados fluem através de um programa de computador, identificando dependências entre variáveis e pontos de acesso aos dados. 5

Análise de Pontos de Programação É uma técnica de análise estática que consiste em identificar os pontos críticos de um programa de computador, como *loops*, chamadas a funções externas, entre outros, para avaliar a sua complexidade e eficiência. 5

Análise Léxica É a primeira fase do processo de compilação de um programa de computador, onde se analisa a estrutura de um código fonte para identificar e categorizar os diferentes elementos léxicos. 5

Análise Sintática É a segunda fase do processo de compilação de um programa de computador, onde se verifica se a sequência de *tokens* do código fonte está de acordo com a gramática da linguagem de programação em questão. 5

C É uma linguagem de programação de propósito geral amplamente utilizada para o desenvolvimento de sistemas operacionais, aplicações de sistema e software de baixo nível. Foi desenvolvida originalmente por Dennis Ritchie na década de 1970. 7

Canais Em *UPPAAL*, são mecanismos de comunicação que permitem a troca de informações entre os componentes do sistema, especificando os tipos de mensagens e as restrições de sincronização. 16

Invariantes Em *UPPAAL*, as invariantes são condições que devem ser verdadeiras em todas as localizações do sistema durante a execução, garantindo propriedades específicas do sistema. 5, 6

Localizações Em *UPPAAL*, as localizações representam os estados do sistema, onde as ações ocorrem e as transições podem ocorrer. Cada localização possui um conjunto de invariantes e condições de guarda que determinam sua ativação e desativação. 15, 16, 22, 27, 31, 32, 36, 39, 45, 46, 49, 53, 55

Capítulo

1

Introdução

1.1 Enquadramento

Este projeto é enquadrado na cadeira de Projeto e tem como base temas introduzidos na cadeira de Sistemas Operativos, tudo feito na Universidade da Beira Interior (UBI).

1.2 Motivação

Durante o curso de licenciatura em Engenharia Informática, tive a oportunidade de explorar diversas áreas do conhecimento e desenvolver uma compreensão mais profunda sobre a área escolhida para projeto. Ao longo desses anos de estudo, despertei uma curiosidade por problemas de *Mutual Exclusion* e *Deadlock* que foram introduzidas na cadeira de Sistemas Operativos. A escolha do tema deste projeto final de licenciatura foi motivada pela minha paixão inegável pelo assunto.

1.3 Documentos do projeto

Todos os documentos e códigos associados a este documento encontram-se disponíveis para consulta na plataforma Github `https://github.com/eduardoGONUBI/uppaal-project`.

1.4 Objetivos

Os objetivos deste projeto são uma introdução à plataforma *Uppaal* para Modelagem, Validação e Verificação de sistemas em tempo real. Em particular queremos explorar a implementação com *Uppaal* de sistemas temporais, explorar a verificação de algoritmos de exclusão mútua e verificar cenários de *deadlocks*. Além disso modelar problemas que são normalmente apresentados durante um curso universitário inicial de Sistemas Operativos (SO).

1.5 Organização do Documento

De modo a refletir o trabalho que foi feito, este documento encontra-se estruturado da seguinte forma:

1. O primeiro capítulo – **Introdução** – apresenta o projeto, a motivação para a sua escolha, o enquadramento para o mesmo, os seus objetivos e a respetiva organização do documento.

2. O segundo capítulo– **Estado da Arte** – apresenta um resumo atualizado dos conhecimentos, métodos e avanços relevantes na área de estudo do projeto.

3. O terceiro capítulo– **Tecnologias Utilizadas** – descreve os conceitos mais importantes no âmbito deste projeto, bem como as tecnologias utilizadas durante o desenvolvimento do mesmo.

4. O quarto capítulo – **Algoritmo de Modelação Temporal** – apresenta a implementação dum modelo dum algoritmo dinâmico temporal de um parque de estacionamento.

5. O quinto item – **Algoritmia de Programação Concorrente** – apresenta a implementação para modelar vários algoritmos de exclusão mútua e *deadlocks*.

6. O sexto capítulo – **Conclusões e Trabalho Futuro** – é apresentado algumas limitações e alguns avanços com as modelações apresentadas, são referidos algumas breves conclusões e apresentados algumas situações que poderiam ser feitas no futuro.

Capítulo

2

Estado da Arte

2.1 Introdução

Neste capítulo, vamos explorar a área da verificação de *software*, muitas vezes vista como um sub-tema de Engenharia de *Software*. Serão abordados inicialmente temas fundamentais, como os tipos de verificação estática e dinâmica. Além disso, iremos considerar as várias abordagens a verificação formal, como *model checking* (verificação de modelos) e o uso de verificação dedutiva ou inferência lógica, como exemplo usamos a ferramenta *Why3*. Depois discutiremos vários *model checkers*, em detalhe.

2.2 Verificação de Software

A verificação de *software* é um processo essencial na garantia de qualidade e confiabilidade de sistemas de *software*. Consiste em avaliar se um programa ou sistema atende a determinadas especificações, identificando possíveis erros, falhas de projeto ou violações de propriedades desejadas. Através desse processo, é possível minimizar riscos e melhorar a segurança e eficiência dos *softwares* desenvolvidos [1].

A verificação de *software* pode ser realizada de diferentes maneiras, mas as duas abordagens principais são: a verificação estática e a verificação dinâmica [2].

Ao realizar a verificação de *software*, é importante considerar diversos aspetos, como a *Correctness* funcional, a segurança, a eficiência e a robustez do sistema. Isso envolve a definição das propriedades a serem verificadas, a criação de casos de teste abrangentes e a aplicação de técnicas de análise adequa-

das. É sempre recomendável combinar diferentes abordagens e ferramentas para obter uma análise completa e eficaz do *software* [3].

2.2.1 Verificação de *Software* Dinâmica

A verificação de *software* dinâmica é uma abordagem para avaliar o comportamento de um programa durante a sua execução. Diferente da verificação estática, que se baseia na análise do código-fonte, a verificação dinâmica concentra-se no comportamento real do *software*, permitindo identificar erros, falhas e violações de propriedades durante a execução do programa [4].

A principal finalidade da verificação de *software* dinâmica é garantir que o programa comporta-se de acordo com as especificações e requisitos definidos. Ela envolve a execução de casos de teste e a monitorização do comportamento do *software* em tempo de execução. Essa abordagem permite identificar problemas como erros de lógica, falhas de memória, condições de corrida e exceções não tratadas, que podem afetar a *Correctness*, a eficiência e a segurança do *software* [5].

Um exemplo comum da verificação dinâmica é a execução de testes automatizados. Nesse caso, são criados casos de teste que representam diferentes cenários de uso do *software*. Ao executar esses casos de teste, é possível observar o comportamento do *software* e verificar se ele produz os resultados esperados e se lida corretamente com situações inesperadas.

Uma das principais vantagens da verificação de *software* dinâmica é a capacidade de identificar erros e falhas que podem passar despercebidos na verificação estática.

No entanto, a verificação dinâmica também apresenta algumas desvantagens. Em primeiro lugar, pode ser difícil cobrir todos os casos de teste, especialmente em sistemas complexos. Além disso, a execução de testes pode ser demorada e exigir recursos computacionais significativos. Outra desvantagem é que esta geralmente identifica problemas após a ocorrência dos mesmos, o que pode afetar a confiabilidade do *software* em ambientes críticos [6].

2.2.2 Verificação de *Software* Estática

A verificação estática é uma abordagem para avaliar a qualidade e confiabilidade de um programa sem a necessidade de executá-lo. Ao contrário da verificação dinâmica, que se concentra no comportamento em tempo de execução, a verificação estática analisa o código-fonte ou alguma representação intermediária para identificar potenciais erros, falhas e violações de propriedades [7].

A principal finalidade da verificação estática é detectar problemas no código antes da execução. Essa abordagem envolve a aplicação de técnicas e ferramentas de análise estática, como Análise Léxica, Análise Sintática, Análise de Fluxo de Dados e Análise de Pontos de Programação, para identificar erros de sintaxe, problemas de estilo, violações de boas práticas de programação e possíveis *bugs* [8].

Um exemplo comum disso é a utilização de ferramentas de análise de código-fonte, que examinam o código em busca de padrões conhecidos de erros e violações de estilo. Essas ferramentas podem identificar problemas como variáveis não inicializadas, acesso incorreto à memória, operações inseguras de entrada e saída, entre outros. A verificação estática também pode incluir a revisão manual do código por parte de programadores ou especialistas em busca de erros e problemas específicos.

Além disso, pode ajudar a melhorar a legibilidade, a manutenção e a escalabilidade do código, uma vez que cria questões relacionadas à estrutura, à organização e à qualidade do *software*.

No entanto, a verificação estática também apresenta algumas desvantagens. Por ser uma análise baseada em um código-fonte ou outra representação, pode não capturar todos os problemas ou situações específicas que podem ocorrer durante a execução. Além disso, algumas técnicas de verificação estática podem gerar falsos positivos ou falsos negativos, ou seja, identificar problemas inexistentes ou não identificar problemas reais, respetivamente [9].

2.2.3 Tipos de lógica utilizadas na verificação de software

Uma das bases fundamentais da verificação de *software* é a utilização de lógica formal para especificar e analisar propriedades dos sistemas. Existem vários tipos de lógica que desempenham papéis importantes nesse contexto. Vamos apenas mencionar brevemente alguns dos principais tipos de lógica utilizados na verificação de *software*.

Lógica Proposicional: A lógica proposicional é uma forma de lógica que lida com proposições e conecta as mesmas através de operadores lógicos, como conjunção, disjunção e negação. Na verificação de *software*, a lógica proposicional é usada para expressar propriedades sobre o comportamento do sistema, como Invariantes e restrições de estado.

Lógica de Predicados: A lógica de predicados é uma extensão da lógica proposicional que permite a quantificação sobre variáveis e predicados. Ela permite expressar propriedades mais complexas sobre o comportamento do sistema, incluindo a especificação de pré-condições e pós-condições de funções e procedimentos.

Lógica Temporal: A lógica temporal é uma forma de lógica que permite expressar propriedades ao longo do tempo. Ela é usada na verificação de *software* para descrever comportamentos dinâmicos, como sequências de eventos, ordem de execução e propriedades de tempo, como a ocorrência de eventos em uma determinada sequência.

Lógica de Primeira Ordem: A lógica de primeira ordem é uma extensão da lógica de predicados que permite a quantificação sobre conjuntos de elementos e a expressão de propriedades mais complexas. Ela é usada na verificação de *software* para especificar propriedades sobre estruturas de dados complexas, como listas, árvores e grafos.

A utilização de lógica na verificação de software permite uma abordagem rigorosa e formal para especificar propriedades e realizar análises precisas sobre o comportamento dos programas.

2.2.4 *Hoare Logic*

É uma técnica matemática que desempenha um papel crucial na área da verificação de *software*. Criada pelo cientista Tony Hoare na década de 1960, a *Hoare Logic* fornece uma estrutura formal para especificar e verificar a *Correctness* de programas [10].

A principal finalidade é estabelecer uma base lógica para a verificação formal de programas. Ela permite especificar propriedades importantes do programa em termos de pré-condições, pós-condições e Invariantes, e verificar se essas propriedades são preservadas durante a execução do programa.

A *Hoare Logic* é baseada em regras de inferência, que definem como as pré-condições e pós-condições são relacionadas em diferentes pontos de um programa. Essas regras estabelecem uma sequência lógica de passos que permitem deduzir a *Correctness* do programa [11].

Ao utilizar esta lógica, é possível expressar formalmente as condições que devem ser satisfeitas antes e após a execução de cada instrução ou bloco de código. Isso facilita a verificação de propriedades como a correta manipulação de variáveis, a ausência de erros, a preservação de Invariantes e a correta utilização de recursos.

2.2.5 *Why3*

O *Why3* é uma ferramenta poderosa e versátil para a verificação de *software*. Projetado para garantir a *Correctness* dos programas, o *Why3* oferece um ambiente de verificação formal, permitindo especificar e verificar propriedades importantes dos sistemas.

Desempenha um papel fundamental ao auxiliar os programadores a criar programas robustos e confiáveis. Ele suporta várias linguagens de programação, incluindo C, *JAVA* e *OCaml*, e oferece uma ampla gama de técnicas e ferramentas para análise e verificação.

Uma das suas principais vantagens é a capacidade de especificar propriedades e realizar verificações formais de maneira sistemática. Permite expressar propriedades lógicas e restrições sobre o comportamento dos programas, garantindo que eles atendam aos requisitos de correção e segurança.

O *Why3* utiliza uma linguagem de especificação, chamada de *WhyML*. Esta permite descrever formalmente as características do programa e as suas propriedades.

Além disso, oferece suporte a diferentes provadores de teoremas, como *Z3*, *Alt-Ergo* e *CVC4*, permitindo a realização de provas automáticas e a verificação de propriedades complexas. Isso ajuda os programadores a detetar erros, violações de contratos e garantir a correção dos programas.

Outra vantagem do *Why3* é a capacidade de gerar código confiável a partir das especificações verificadas. Isso possibilita a geração automática de código em diferentes linguagens, o que reduz os erros humanos e aumenta a confiabilidade dos sistemas desenvolvidos.

2.2.6 Teorema do Arroz

O Teorema do Arroz (do matemático americano Henry Gordon Rice) é um resultado importante da teoria da computabilidade que tem implicações relevantes na verificação de *software*. Nesta, uma das abordagens é a especificação semântica, que consiste em definir um conjunto de execuções corretas de um programa.

Existem especificações semânticas triviais, como o conjunto vazio (nenhuma execução é correta) ou o conjunto de todas as execuções possíveis. Intuitivamente, estas especificações triviais não são interessantes para a verificação de *software*, pois não fornecem informações úteis sobre o comportamento do programa.

Um sistema *Turing Complete* significa um sistema no qual um programa pode ser escrito que encontrará uma resposta (embora sem garantias quanto ao tempo de execução ou memória). Neste contexto, O Teorema do Arroz, formulado em 1953, afirma que, considerando uma linguagem de programação *Turing* completa, qualquer especificação semântica não trivial não é computável.

Isso significa que propriedades interessantes, como a terminação do programa, a ausência de erros em tempo de execução ou a ausência de erros aritméticos, são intrinsecamente não computáveis. O Teorema do Arroz mostra-

nos que não podemos ter uma solução geral para esses problemas de verificação de *software*.

No entanto ainda podemos resolver problemas mais fracos e fazer alguns compromissos. Por exemplo, podemos realizar simulações ou testes observando apenas um número finito de execuções do sistema, podemos utilizar provadores de teoremas assistidos, podemos utilizar a técnica de *model checking* considerando apenas sistemas finitos, ou podemos procurar por "padrões"que indiquem possíveis erros no código. [12]

Essas abordagens parciais permitem-nos lidar com a verificação de sistemas em diferentes graus de automação, escalabilidade e abrangência.

2.3 *Model Checkers*

Na área da verificação de sistemas, uma das ferramentas mais valiosas e poderosas são os *model checkers*. Mas afinal, o que são e qual é a sua utilidade?

Os *model checkers* são ferramentas automatizadas que permitem analisar de forma sistemática e exaustiva todas as possíveis configurações e comportamentos de um sistema, em busca de erros e violações de propriedades especificadas.

A sua utilidade é vasta, uma vez que os *model checkers* podem ajudar a garantir a correção e confiabilidade de sistemas complexos, como sistemas embarcados, protocolos de comunicação, circuitos eletrónicos, entre outros. Ao verificar todas as combinações de estados possíveis de um sistema, os *model checkers* conseguem detetar problemas que poderiam passar despercebidos em testes tradicionais ou análise manual.

Uma das suas grandes vantagens é a capacidade de explorar todas as trajetórias possíveis de um sistema, o que permite identificar erros e propriedades violadas que poderiam ocorrer em situações pouco prováveis, mas ainda assim relevantes. Essas ferramentas podem ajudar a encontrar problemas como *deadlocks*, condições de corrida, violações de segurança, entre outros.

Além disso, oferecem um processo de verificação automatizado, o que facilita a deteção de erros e torna o processo de verificação mais eficiente. Eles realizam a análise de forma completa e sistemática, permitindo que os programadores concentrem os seus esforços na correção dos problemas identificados.

Existem vários model checkers disponíveis, cada um com as suas características e áreas de aplicação específicas. Alguns exemplos populares incluem o *SPIN*, o *Uppaal* e o *TLA+*.

É importante ressaltar que os *model checkers* não substituem outras técnicas de verificação, como testes unitários, análise estática e revisões de código.

Pelo contrário, eles complementam essas abordagens, oferecendo uma verificação adicional e mais abrangente[13].

2.3.1 *SPIN*

O *SPIN* é um *model checker* amplamente utilizado na verificação de sistemas concorrentes. Esta ferramenta realiza a verificação formal de programas, protocolos de comunicação e sistemas distribuídos, detetando possíveis erros e violações de propriedades especificadas. Vamos explorar o que é o *SPIN*, para que serve e as suas vantagens e desvantagens.

Utiliza uma abordagem de modelação baseada em autómatos de estados finitos, onde os estados representam configurações do sistema e as transições entre os estados representam as mudanças de estados possíveis.

Uma das principais utilidades do *SPIN* é a verificação de sistemas concorrentes, nos quais múltiplos processos são executados de forma simultânea e interagem entre si. Ele permite especificar a estrutura do sistema, incluindo a definição dos processos e a forma como eles se comunicam. Com base nessa especificação, o *SPIN* analisa exaustivamente todas as possíveis interações entre os processos, verificando propriedades como ausência de *Deadlock*, condições de corrida e coerência da comunicação.

Entre as vantagens do *SPIN*, destaca-se a sua capacidade de realizar uma verificação completa e sistemática do sistema em análise. Ele explora todas as possíveis trajetórias de execução, o que possibilita encontrar erros que poderiam passar despercebidos em testes convencionais ou análise manual. Além disso, o *SPIN* é altamente eficiente na verificação de sistemas concorrentes, permitindo lidar com modelos complexos e grandes quantidades de estados.

No entanto, assim como qualquer ferramenta, o *SPIN* também apresenta algumas desvantagens. A primeira delas é a explosão do espaço de estados. Em sistemas com um grande número de processos e variáveis, o número de estados possíveis pode crescer exponencialmente, tornando a verificação impraticável ou consumindo uma quantidade excessiva de recursos computacionais. Além disso, o *SPIN* exige uma especificação precisa do sistema, o que pode exigir um esforço significativo na fase inicial do processo de verificação.

2.3.2 *TLA+*

O *TLA+* é uma ferramenta de verificação que utiliza a lógica temporal para analisar a *Correctness* de sistemas complexos. Desenvolvido por Leslie Lamport, o *TLA+* permite a especificação formal de sistemas e a verificação de propriedades desejadas. Neste subcapítulo, vamos explorar o que é o *TLA+*, para que serve e quais são as suas vantagens e desvantagens.

O *TLA+* é uma linguagem de especificação e um conjunto de ferramentas associadas que permitem descrever sistemas em um nível abstrato. Ele fornece uma linguagem matemática precisa e expressiva para especificar a estrutura, comportamento e restrições de um sistema. Com o *TLA+*, é possível modelar sistemas concorrentes, sistemas distribuídos, protocolos de comunicação e muitos outros sistemas complexos.

A principal utilidade do *TLA+* é a verificação de sistemas críticos, onde a correção e confiabilidade são essenciais. Permite a especificação de propriedades desejadas e a realização de verificações formais para garantir que essas propriedades sejam satisfeitas pelo sistema modelado. Com o *TLA+*, é possível analisar a consistência, a *Correctness* e a segurança de um sistema, identificando possíveis erros e falhas antes mesmo da implementação.

Entre as vantagens do *TLA+*, destaca-se a sua capacidade de lidar com a complexidade de sistemas reais. A linguagem permite uma abstração adequada para representar sistemas com múltiplos componentes, comportamento assíncrono e interações complexas entre os elementos. Além disso, o *TLA+* possui um sólido suporte matemático, fornecendo uma base sólida para a verificação formal e a garantia de propriedades críticas.

Outra vantagem do *TLA+* é a sua extensibilidade e modularidade. Ele permite a especificação de sistemas em diferentes níveis de detalhe, desde uma visão de alto nível até aspetos mais refinados. Além disso, o *TLA+* pode ser integrado com outras ferramentas de verificação e simulação, proporcionando um ambiente completo para a análise formal de sistemas.

Porém, assim como qualquer ferramenta, o *TLA+* também apresenta algumas desvantagens. Uma delas é a curva de aprendizagem inicial, pois a linguagem e a abordagem podem exigir tempo e esforço para serem dominadas. Além disso, a verificação formal com o *TLA+* pode ser computacionalmente intensiva, especialmente para sistemas complexos com um grande número de estados possíveis.

2.3.3 *Uppaal*

O *Uppaal* será introduzido num capítulo posterior visto que este foi o *model checker* escolhido para este projeto.

2.3.4 *Uppaal* vs *SPIN* e *TLA+*

Embora cada uma dessas ferramentas tenha as suas próprias características e áreas de aplicação, o *Uppaal* destaca-se em certos aspetos, oferecendo vantagens distintas em comparação com o *TLA+* e o *SPIN*.

Uma das principais vantagens do *Uppaal* é a sua capacidade de lidar com sistemas de tempo real. Ele foi especialmente projetado para modelar e verificar sistemas em que o tempo é uma dimensão crítica, como sistemas embarcados e protocolos de comunicação em tempo real. O *Uppaal* permite a especificação precisa de restrições temporais e o comportamento concorrente, garantindo que as propriedades relacionadas ao tempo sejam devidamente verificadas.

Outra vantagem do *Uppaal* é a sua interface gráfica intuitiva e amigável. A ferramenta oferece um ambiente visual que facilita a modelação e a visualização de sistemas, permitindo aos utilizadores construir modelos de forma mais eficiente. A interface gráfica do *Uppaal* permite a representação de diagramas de estados, gráficos temporais e outras visualizações relevantes, tornando a verificação mais compreensível e facilitando a deteção de erros.

Tendo em conta estas vantagens, este foi escolhido para o desenvolvimento deste projeto, visto que este permite uma melhor modelação dos algoritmos propostos.

2.3.5 Redes de Petri

Para terminar este breve resumo da área da verificação de sistemas, acabaremos com as Redes de *Petri*. Estas são uma ferramenta utilizada na modelação e análise de sistemas concorrentes. Elas fornecem uma linguagem de modelação usando uma representação gráfica intuitiva que permite descrever o comportamento de sistemas distribuídos, paralelos e assíncronos [14].

Consistem em um grafo direcionado, no qual os lugares representam estados do sistema e as transições representam as ações que podem ocorrer. O fluxo de *tokens* entre os lugares e as transições representa a evolução do sistema ao longo do tempo. Essa representação gráfica torna as mesmas numa ferramenta visual poderosa para modelar e compreender o comportamento de sistemas complexos.

Elas permitem representar formalmente o comportamento do sistema, incluindo a concorrência, as interações entre os componentes e as restrições temporais.

A relação entre *model checkers* e redes de *Petri* reside no fato de que estas podem servir como base para criar modelos que podem ser analisados usando técnicas de *model checkers*. Através da análise desses modelos, os *model checkers* podem verificar a presença de erros, propriedades indesejadas ou inconsistências, alcançabilidade, segurança, vivacidade, ausência de bloqueios e outras especificações de lógica temporal [13].

2.4 Conclusões

Com isto é encerrado este capítulo onde exploramos a fascinante área da verificação de *software*, discutimos os tipos de verificação estática e dinâmica, a lógica de *Hoare*, os tipos lógicos. Foi também apresentado o *Why3* e diversos *model checkers*, ferramentas essenciais para a verificação de *software*. Com essas técnicas e ferramentas, podemos garantir a qualidade, confiança e segurança dos programas desenvolvidos.

Capítulo

3

Tecnologias e Ferramentas Utilizadas

3.1 Introdução

Neste capitulo serão introduzidas as duas ferramentas utilizadas no desenvolvimento do projeto e na escrita do relatório do mesmo. Estas são o *Uppaal* e o *Overleaf*. Como neste projeto o foco principal é na utilização da ferramenta *Uppaal* farei uma breve descrição do *Overleaf* seguido por uma descrição detalhada do *Uppaal*.

3.2 *Overleaf*

O *Overleaf* é uma plataforma online de edição de documentos *LaTeX*, que permite criar, editar e partilhar documentos científicos, académicos e técnicos. *LaTeX* é uma linguagem de marcação utilizada para produzir documentos com alta qualidade tipográfica e é amplamente utilizado na comunidade académica e científica.

No *Overleaf*, é possível escrever um documento *LaTeX* a partir de modelos pré-definidos ou criar um documento do zero. A plataforma oferece uma interface intuitiva e fácil de usar. Além disso, o *Overleaf* possui recursos que facilitam o trabalho em equipa e a colaboração em tempo real.

Figura 3.1: Logótipo do *Overleaf*.

3.3 *Uppaal*

O *Uppaal* é uma poderosa ferramenta de modelação e verificação de sistemas concorrentes em tempo real. Desenvolvido por investigadores da Universidade de Uppsala e da Universidade de Aalborg, o *Uppaal* combina a teoria dos autómatos temporizados com a lógica temporal, oferecendo recursos avançados para a análise rigorosa e a verificação de propriedades de sistemas complexos.

Ele emprega a notação de Autómatos Temporizados, que adicionam informações de tempo aos autómatos tradicionais, permitindo a representação precisa de sistemas em tempo real. Com essa notação, é possível especificar as transições entre estados, as ações dos componentes do sistema e as restrições de tempo que governam o comportamento do sistema.

Além da modelação, o *Uppaal* oferece recursos de verificação.

A ferramenta utiliza a linguagem *Timed Computation Tree Logic* (TCTL) para especificar propriedades temporais e consultas sobre o modelo. Com a TCTL, é possível verificar se determinadas propriedades desejadas são satisfeitas pelo sistema modelado.

Uma das principais vantagens do *Uppaal* é a capacidade de realizar verificações automáticas em modelos complexos. A ferramenta emprega técnicas avançadas, como simulação de execução, geração de grafos de estados e técnicas de *model checking*, para analisar o comportamento do sistema e identificar possíveis problemas.

No entanto, é importante destacar que o *Uppaal* também apresenta desvantagens como a escalabilidade em sistemas de grande porte, onde a análise pode se tornar computacionalmente intensiva.

A seguir, estão de forma resumida as formas como o verificador do *Uppaal* entende a linguagem:

- 1-E (*Exists*): A notação "E"é usada para expressar a existência de um caminho no sistema em que uma determinada propriedade é satisfeita. Por exemplo, a expressão "E p"significa que existe um caminho em que a propriedade "p"é verdadeira.

- 2-A (*For All*): A notação "A"é usada para expressar que uma determinada propriedade é verdadeira para todos os caminhos possíveis no sistema. Por exemplo, a expressão "A p"significa que a propriedade "p"é verdadeira em todos os caminhos.

- 3-G ([] - *Globally*): A notação "[]"é usada para expressar que uma propriedade é verdadeira em todos os estados de um caminho. Essa notação representa o comportamento global do sistema. Por exemplo, a

expressão "[] p"significa que a propriedade "p"é verdadeira em todos os estados de todos os caminhos

- 4-F (<>) - *Finally*): A notação «>"é usada para expressar que uma propriedade é verdadeira em pelo menos um estado de um caminho. Essa notação representa o comportamento eventual do sistema. Por exemplo, a expressão «>(p)"significa que a propriedade "p"é verdadeira em pelo menos um estado de algum caminho.

Estas foram baseadas no seguinte vídeo: `https://www.youtube.com/watch?v=r8ydiD4ngGY`. Mais exemplos reais do TCTL utilizado na verificação de um modelo não definido:

- E<> p: existe um caminho onde p eventualmente se mantém.
- A[]p: Para todos os caminhos, p sempre vale.
- E[]p: Existe um caminho onde p sempre ocorre.
- A<> p: para todos os caminhos, p eventualmente será válido.
- p –> q: sempre que p for válido, q eventualmente será válido.

Esta ferramenta foi usada para modelar e resolver vários algoritmos de *Mutual Exclusion* e *Deadlock* que eram o propósito do projeto.

Figura 3.2: Logótipo do *Uppaal*

3.3.1 Como se modela em *Uppaal*

Ao modelar um problema no *Uppaal*, é importante seguir algumas etapas para garantir uma representação precisa e eficaz do sistema. Aqui seguem alguns passos básicos para modelar um problema no *Uppaal*:

- Identificar os componentes do sistema: Começa-se identificando os componentes ou entidades envolvidas no problema.
- Definir as Localizações: Identificar os estados possíveis em que o sistema pode-se encontrar e representar os mesmos como Localizações. Dar nomes significativos a cada Localizações facilita a compreensão do modelo.

- Criar as *Edges*: Descrever as transições entre as Localizações por meio de *Edges*. Essas *Edges* representam as ações que levam o sistema de um estado para outro. Associar *Guards* a cada aresta para indicar quando a transição pode ocorrer.

- Adicionar variáveis: Caso o problema envolva informações dinâmicas ou variáveis que mudam ao longo do tempo, adicionar variáveis ao modelo a desenvolver. Essas variáveis podem representar quantidades, estados ou qualquer outra informação relevante para o sistema.

- Integrar restrições temporais: Se houver restrições de tempo no seu problema, utilizar relógios para representar as mesmas. Os relógios ajudam a definir restrições temporais e eventos que ocorrem em momentos específicos.

- Sincronizar a comunicação: Se o sistema envolver a comunicação ou interação entre componentes, utilizar Canais para sincronizar essa comunicação. Os Canais permitem a troca de informações entre processos ou entidades e ajudam a coordenar suas ações.

- Especificar propriedades: Identificar as propriedades a verificar no sistema e especificar as mesmas usando a linguagem TCTL, explicada acima.

- Verificar o modelo: Utilizar as ferramentas de verificação do *Uppaal* para analisar o modelo e verificar se as propriedades especificadas são satisfeitas.

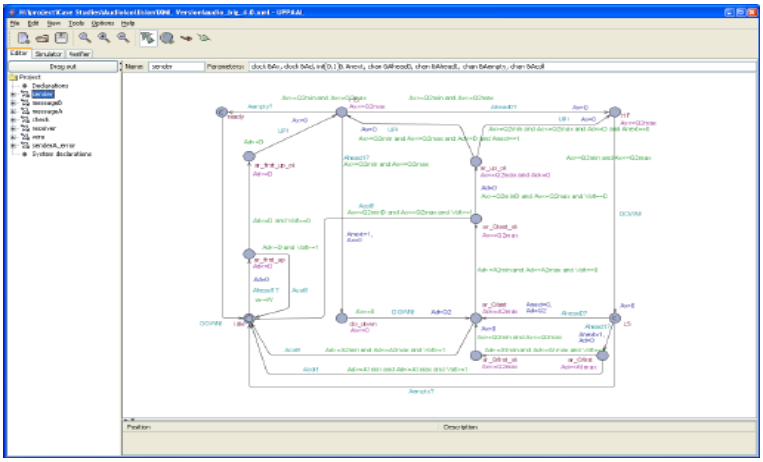

Figura 3.3: Exemplo de um modelo feito pelos criadores do *Uppaal*.

Seguindo estes passos é possível detalhar um modelo em *Uppaal* de forma completa. Por experiência própria, recomendo começar com o algoritmo de Peterson e um parque de estacionamento ou máquina de vendas, visto que é algo fácil e lógico de perceber.

3.3.2 Instalar *Uppaal*

A instalação do *Uppaal* em todas as plataformas é feita das seguintes maneiras:

Windows

Para instalar o *Uppaal* no Windows, siga os passos abaixo:

1. Certifique-se de ter uma versão recente do Java instalada no seu sistema.
2. Verifique que possui privilégios de administrador na máquina.
3. Aceda ao site oficial do *Uppaal* em `http://www.Uppaal.org` e vá para a secção de downloads.
4. Escolha a versão mais recente do *Uppaal* adequada para o seu sistema operacional Windows e faça o download do arquivo de instalação.
5. Após o download, execute o arquivo de instalação e siga as instruções do assistente de instalação.
6. Uma vez concluída a instalação, pode iniciar o *Uppaal* a partir do menu Iniciar ou do atalho na área de trabalho.

Linux

Para instalar o *Uppaal* no Linux, siga as instruções abaixo:

1. Certifique-se de ter uma versão recente do *JAVA* instalada no seu sistema.
2. Abra o terminal e navegue até o diretório onde deseja instalar o *Uppaal*.
3. Aceda ao site oficial do *Uppaal* em `http://www.Uppaal`.org e vá para a seção de downloads.
4. Escolha a versão mais recente do *Uppaal* adequada para o Linux e faça o download do arquivo compactado.

5. Extraia o conteúdo do arquivo compactado para o diretório de instalação.
6. Defina as permissões corretas para os arquivos do *Uppaal*, se necessário, usando o comando `chmod`.
7. Abra o terminal e navegue até o diretório de instalação do *Uppaal*.
8. Execute o comando `./Uppaal` para iniciar o *Uppaal*.

Mac

Para instalar o *Uppaal* no Mac, siga os passos abaixo:

1. Certifique-se de ter uma versão recente do *Java Development Kit* (JDK) instalada no seu sistema.
2. Aceda ao site oficial do *Uppaal* em `http://www.Uppaal.org` e vá para a secção de downloads.
3. Escolha a versão mais recente do *Uppaal* adequada para o Mac e faça o download do arquivo de instalação.
4. Após o download, abra o arquivo de instalação.
5. Arraste o ícone do *Uppaal* para a pasta de aplicativos para concluir a instalação.
6. Agora pode iniciar o *Uppaal* a partir do *Launchpad* ou da pasta de aplicações.

3.4 Conclusões

Com base no que foi apresentado, é possível perceber-se que este projeto foca-se inteiramente no uso e exploração da ferramenta de trabalho *Uppaal* e na introdução da mesma a mais pessoas.

Capítulo

4

Algoritmo de Modelação Temporal

4.1 Introdução

No capítulo seguinte, iremos adentrar no campo da modelação temporal, um importante algoritmo utilizado para aprender melhor como representar o comportamento dinâmico de sistemas, perceber melhor o *Uppaal* e como utilizar o mesmo.

Para isto fizemos a modelação de um parque de estacionamento baseado num modelo demonstrado em 2017 por Maria Bianca uma ex-estudante romena que se mudou para a Suécia em busca de um aprofundamento dos seus conhecimentos. (O modelo pode ser visto na *Uniformed Resource Locator* (URL) `https://www.youtube.com/watch?v=0ioBpqDG0f0`.

De facto este modelo foi também estudado e apresentado na conferência *International Conference on Software Engineering Advances* (ICSEA), 2019 pelos autores Bukhari e Sultana [15]. Foi decidido modificar o modelo de forma a tornar-se mais realista e comparável a um estacionamento português.

4.2 Estacionamento de Carros baseado em Portugal

Sistemas de estacionamento de carros são sistemas interessantes para testar a utilidade de uma ferramenta como o *Uppaal*. O estacionamento de carros torna-se uma atividade cada vez mais problemática em grandes cidades, muitas das quais, por razões ambientais, começam a reduzir o estacionamento disponível para o publico.

Portanto, o objetivo geral deste capítulo é modelar um sistema de estacionamento de carros utilizando o *Uppaal*, a fim de garantir que o sistema seja correto, eficiente e eficaz, o que pode levar a uma melhor compreensão do sistema. Modelamos uma situação do mundo real frequentemente encontrada de uma única faixa com dois ou mais carros e um número flexível de vagas de estacionamento.

A faixa contém um número de vagas de estacionamento. O objetivo é garantir que os dois carros possam percorrer a faixa sem colidir um com o outro, enquanto estes procuram por um lugar de estacionamento livre, é permitido aos carros entrar e sair dos estacionamentos com segurança, evitando colisões.

Para isso um dos carros precisa estar posicionado atrás do outro quando eles percorrem a faixa, não permitimos ultrapassagens a não ser que um dos carros esteja estacionado.

4.3 Críticas ao modelo de Maria Bianca

Ao contrário do nosso modelo, o de Maria consistia em duas faixas e quatro estacionamentos, dois do lado da faixa direita e dois do lado da faixa esquerda, em que os carros vinham lado a lado na mesma direção e procuravam por um estacionamento livre.

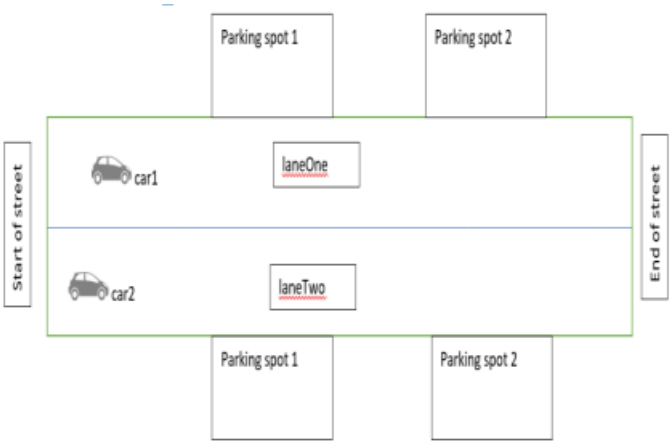

Figura 4.1: Modelo seguido por Maria Bianca ao modelar o problema.

Com este modelo surgem grandes problemas na aplicação ao mundo real.

- Primeiro- O modelo permite que o carro da faixa esquerda estacione nos estacionamentos da faixa direita, sem haver verificações de que o

carro da faixa direita está exatamente do lado deste. Numa situação real haveria uma colisão por causa disto.

- Segundo- O modelo força que ao escolher estacionar do lado direito com um certo carro, o outro carro seja obrigado a estacionar do lado esquerdo mesmo que ainda exista um espaço livre no lado direito.

- Terceiro- O modelo permite o carro da faixa esquerda caso seja estacionado na faixa direita, sair do estacionamento mesmo que o outro carro se encontre exatamente à saída do estacionamento, provocando assim um acidente.

- Quarto- O modelo não permite voltar ao inicio da rua caso queira.

- Quinto- O modelo entra em *deadlock* quando ambos os carros chegam ao fim da rua.

- Sexto- O modelo não é representativo das ruas em Portugal, talvez as ruas na Suécia sejam assim!

Com base nestas criticas o nosso modelo foi modificado e criado para ser mais eficiente e real do que o de Maria Bianca.

4.4 Visão Geral do Sistema

Foi projetado um autómato para representar a faixa por onde viajam os dois carros enquanto procuram por lugares de estacionamento. Os dois carros movem-se na mesma direção. À medida que avançam ao longo da pista, os carros examinam continuamente as vagas de estacionamento para encontrar um espaço disponível. Se uma vaga de estacionamento vazia for detetada, o carro da frente da fila sinalizará ao carro que vem atrás que pretende estacionar, permitindo que o segundo carro se mover para o espaço atrás, uma vez que o carro da frente estacionou. Depois que um carro estaciona, o carro fica no estacionamento por um valor definido de tempo (*clock*) após o qual ele pode sair do estacionamento e voltar para a faixa. É importante observar que os dois carros não podem ocupar a mesma vaga de estacionamento simultaneamente. Quando um carro chega ao final da faixa sem encontrar um espaço livre este pode voltar de novo ao inicio da rua, dando a volta.

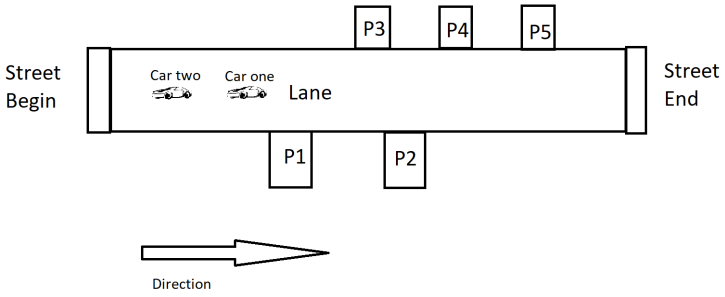

Figura 4.2: Modelo criado para resolver o problema numa situação mais realista.

4.5 Modelo em *Uppaal*

Para modelar isto utilizando o *Uppaal* foram a uma única *template* adicionadas 18 Localizações, todas unidas por *Edges*.

Cinco das Localizações representam os estacionamentos e as treze restantes representam posições intermédias da rua em que não existe estacionamento.

As *Edges* contêm *Guards* e *Updates*/Atualizações que utilizam um *array de booleans* para verificar se é possível avançar com o carro para a posição seguinte e caso isto seja possível registar essa posição como ocupada.

Isto pode ser visualizado na imagem seguinte:

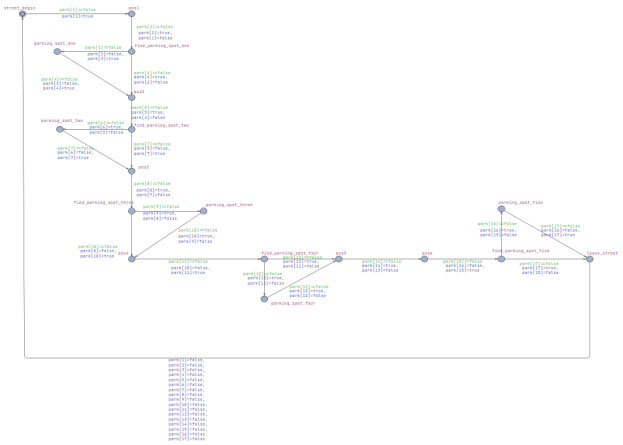

Figura 4.3: Modelo criado por nós do parque de estacionamento em *Uppaal*

4.6 Verificações de Sistema

O sistema foi verificado usando o verificador do *Uppaal* para garantir a segurança, acessibilidade, vivacidade, utilidade e impasses do sistema, como pode ser visto na imagem seguinte.

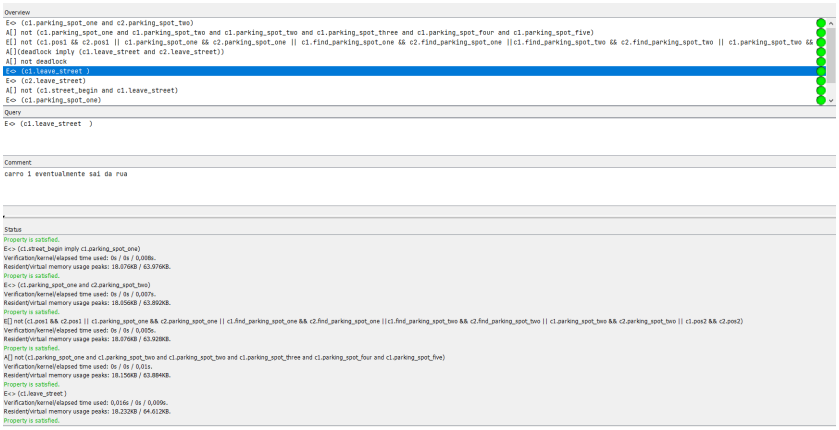

Figura 4.4: Prova de que as verificações foram aprovadas.

4.6.1 *Safety*

- $E <> \neg(c1.parking\_spot\_one \wedge c2.parking\_spot\_one)$: Não existe nenhuma situação em que o carro um e o carro dois estejam estacionados no mesmo sitio ao mesmo tempo.
- $E <> \neg(c1.parking\_spot\_one \wedge c1.parking\_spot\_two)$: Verifica que não existe nenhuma situação em que o carro um está estacionado em vários sítios ao mesmo tempo.

4.6.2 *Reachability*

- $E <> (c1.street\_leave)$: Verifica que o carro um eventualmente sai da rua.
- $E <> (c2.street\_leave)$: Verifica que o carro dois eventualmente sai da rua.

4.6.3 *Utility*

- $E <> (c1.street\_begin \Rightarrow c1.parking\_spot\_one)$: Verifica que quando o carro um entra na rua, este eventualmente vai estacionar o carro.

- $E <> (c2.street\_begin \Rightarrow c2.parking\_spot\_one)$: Verifica que quando o carro dois entra na rua, este eventualmente vai estacionar o carro.

4.6.4 *Mutual Exclusion*

- $A[] \neg (c1.street\_begin \wedge c1.leave\_street)$: Verifica que o carro um não pode estar no inicio e no fim da rua ao mesmo tempo, o mesmo vale para o carro dois.

4.6.5 *Deadlock*

- $A[]((deadlock \rightarrow (c1.leave\_street \wedge c2.leave\_street)))$: Verifica se existe *Deadlock* quando ambos os carros saem da rua.
- $A[] \neg deadlock$: Verifica se existe algum *Deadlock* na totalidade do modelo.

4.7 Conclusões

Ao modelar o parque de estacionamento, pudemos compreender a importância da modelação temporal na verificação de propriedades e no planeamento eficiente de recursos. Identificamos desafios específicos associados à realidade de um estacionamento português e destacamos a relevância de considerar aspetos como gestão de vagas e tempos de permanência.

Ao finalizar este capítulo, percebemos a vastidão das possibilidades oferecidas pela modelação temporal.

Capítulo

5

Algoritmia de Programação Concorrente

5.1 Introdução

Este capítulo apresenta a resolução de problemas de *Mutual Exclusion* e *Deadlock*, crucial para garantir a correta execução de sistemas concorrentes, utilizando o *Uppaal*. Com o mesmo vamos encontrar soluções para os problemas de forma a garantir um comportamento correto e eficiente do sistema.

Para apresentar os vários algoritmos estudados seguimos o raciocínio do livro [16] e a ordem apresentado na disciplina de SO.

5.2 Algoritmo de Alternância Estrita

A alternância estrita é um algoritmo simples de *Mutual Exclusion* para dois processos. Ela garante que os processos alternem para aceder a uma secção crítica, permitindo que apenas um deles aceda de cada vez. O algoritmo utiliza uma variável compartilhada que indica qual processo tem o direito de aceder a secção crítica, alternando o seu valor entre os processos de forma estrita. Dessa forma, a *Mutual Exclusion* é garantida, evitando condições de disputa e conflitos de recursos.

O algoritmo obedece a algumas condições, estas são:

- Garante a *Mutual Exclusion*, mas só é aplicável a 2 processos.
- Não garante a progressão. Se um dos processos quiser entrar 2 vezes consecutivas na secção critica, não pode. Só há a garantia da alternância dos processos na execução da secção crítica.

- Não garante a espera limitada. Se um processo falha o outro ficará permanentemente bloqueado.

- Só funciona se os dois processos forem absolutamente alternativos.

Além disto, segue o seguinte algoritmo:

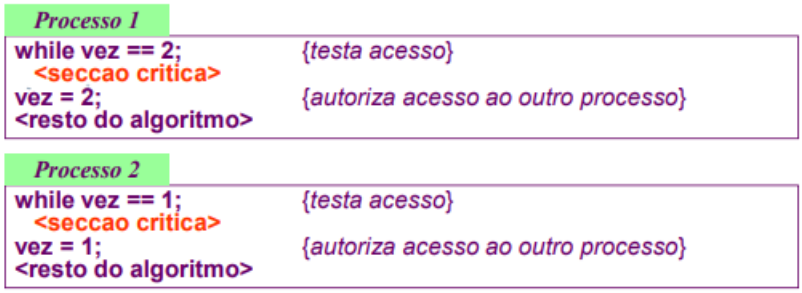

Figura 5.1: Algoritmo seguido pela alternância estrita.

5.2.1 Criticas ao algoritmo

A Solução Alternância Estrita, apesar de fornecer uma abordagem simples para garantir a *Mutual Exclusion* entre dois processos, é alvo de algumas críticas. A seguir, destacaremos algumas das críticas mais comuns a esse algoritmo:

- Limitação a dois processos: O algoritmo só é aplicável quando existem exatamente dois processos envolvidos. Essa restrição torna-o inadequado para sistemas com mais de dois processos. Em cenários mais complexos, onde múltiplos processos precisam compartilhar recursos de forma exclusiva, a Solução Alternância Estrita não é uma opção viável.

- Falta de progressão: Um problema significativo desse algoritmo é que ele não garante a progressão dos processos. Se um processo deseja entrar na secção crítica duas vezes consecutivas, ele não será capaz de fazê-lo. Isso pode resultar em subutilização de recursos, pois o processo não pode aceder a secção crítica quando precisa.

- Espera ilimitada: Outra limitação crítica é que o algoritmo não garante uma espera limitada. Se um processo falhar ou ficar bloqueado por qualquer motivo, o outro processo ficará permanentemente bloqueado,

levando a um *deadlock*. Essa situação pode levar a um desempenho insatisfatório e pode exigir uma reinicialização completa do sistema para resolver o impasse.

- Rigidez na alternância: A Solução Alternância Estrita pressupõe que os processos sigam um padrão fixo de alternância (1, 2, 1, 2, etc.). Isso significa que os processos devem ser absolutamente alternativos. No entanto, em sistemas reais, pode ser difícil garantir que os processos sigam um padrão estritamente alternativo. Essa rigidez na alternância pode limitar a flexibilidade e a adaptabilidade do sistema.

- Ineficiência em relação ao desempenho: A Solução Alternância Estrita não é a mais eficiente em termos de desempenho. Uma vez que os processos precisam testar a variável "vez"continuamente para verificar se podem entrar na secção crítica, pode ocorrer um alto *overhead* de comunicação entre os processos. Isso pode levar a uma utilização ineficiente do processador e aumentar o tempo de resposta do sistema como um todo.

Como se pode verificar embora a Solução Alternância Estrita ofereça uma abordagem básica para a *Mutual Exclusion* entre dois processos, ela possui várias limitações e críticas significativas.

5.2.2 Modelar em *Uppaal*

Para modelar este algoritmo em *Uppaal* foram a uma única *template* adicionadas 2 Localizações, uma com o nome de "remainder"(representa o inicio do sistema), outra com o nome "cs"(a secção critica).

Existem 2 *Edges* que consistem no seguinte:

- remainder->cs, com uma *Guards* "turn==me", que verifica se o turno do processo pode aceder à secção critica.

- cs->remainder, com uma *Updates*/Atualizações "turn:=(me==1?2:1)"que modifica o valor do turno para o oposto, por exemplo se o turno atual é 1 vai atualizar para 2, garantindo assim a alternância estrita no acesso à secção critica.

Foi adicionado à template o parâmetro "const int me", para identificar o processo e compará-lo com a variável turn que foi declarada como "int[1,2] turn;"(esta representa a variável vez no pseudocódigo e é relacionado com o "me"dependendo do número do processo).

Isto gera o seguinte modelo no *Uppaal*:

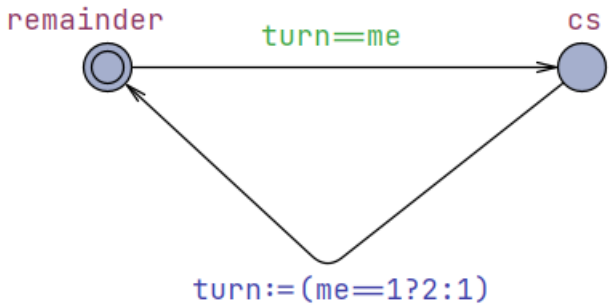

Figura 5.2: Modelo em *Uppaal* do algoritmo de alternância estrita.

5.2.3 Verificações

O sistema foi verificado usando o verificador do *Uppaal* para garantir que este corre como o algoritmo é suposto.

5.2.3.1 *Safety*

Não aplicável.

5.2.3.2 *Mutual Exclusion*

- A[] (P1.cs imply not P2.cs), verifica que se o processo 1 estiver na secção critica o processo 2 não está lá.
- A[] (P2.cs imply not P1.cs), verifica que se o processo 2 estiver na secção critica o processo 1 não está lá.

5.2.3.3 *Deadlock*

- A[] not deadlock

5.2.3.4 *Utility*

- E<> (P2.remainder imply P2.cs), verifica que eventualmente se o processo 2 estiver no remainder ele vai aceder à secção critica.
- E<> (P1.remainder imply P1.cs), verifica que eventualmente se o processo 1 estiver no remainder ele vai aceder à secção critica.

5.2.3.5 *Reachability*

- E<>(P1.cs), verifica que eventualmente o processo 1 estará na secção critica.
- E<>(P2.cs), verifica que eventualmente o processo 2 estará na secção critica.

Pode ser visto o sucesso das verificações na imagem seguinte:

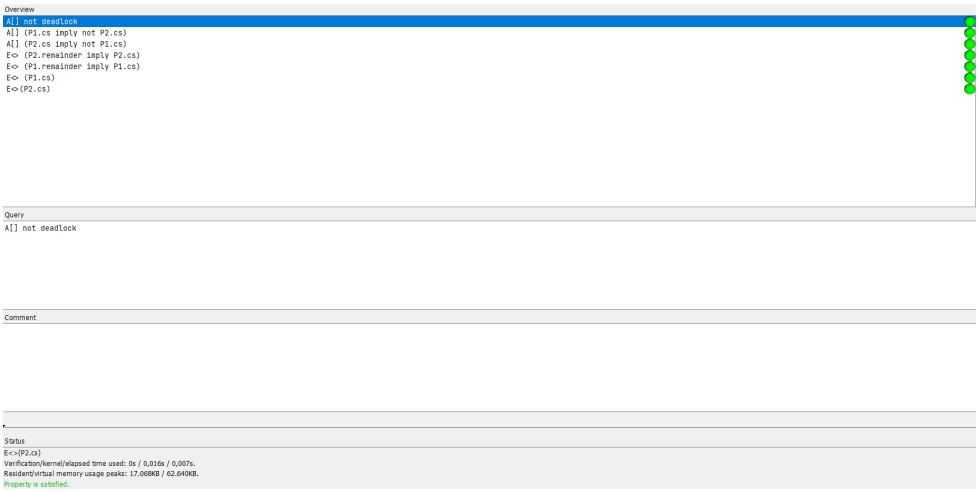

Figura 5.3: Captura de ecrã do verificador do *Uppaal* que mostra o sucesso das propriedades indicadas em cima.

5.3 Algoritmo de Dekker

O algoritmo de Dekker é um algoritmo clássico utilizado para garantir *Mutual Exclusion* entre dois processos ou *threads* concorrentes em um sistema computacional.

Ele foi proposto por Edsger Dijkstra em 1965 e recebeu esse nome em homenagem a seu colega Thomas J. Dekker. Tem uma grande importância histórica visto que foi a primeira solução comprovadamente correta para o problema da *Mutual Exclusion* duma secção crítica.

O objetivo do algoritmo é permitir que dois processos partilhem uma secção crítica do código de forma a que apenas um deles possa aceder a mesma por vez. A ideia básica por trás do algoritmo é a utilização de variáveis booleanas compartilhadas para controlar o acesso à secção crítica.

É baseado em um esquema de rodízio, onde cada processo tem uma vez "turno"para aceder a secção crítica. Para isso, cada processo alterna entre

duas fases: "entrar"e "esperar". A fase "entrar"é onde o processo solicita o acesso à secção crítica, enquanto a fase "esperar"é onde o processo espera pacientemente até que seja a sua vez de entrar.

O algoritmo obedece a algumas condições, estas são:

- Garante *Mutual Exclusion*, mas obriga a que o número de processos seja 2.
- Garante a progressão.
- Não obriga à alternância estrita.

O algoritmo é o seguinte 5.4:

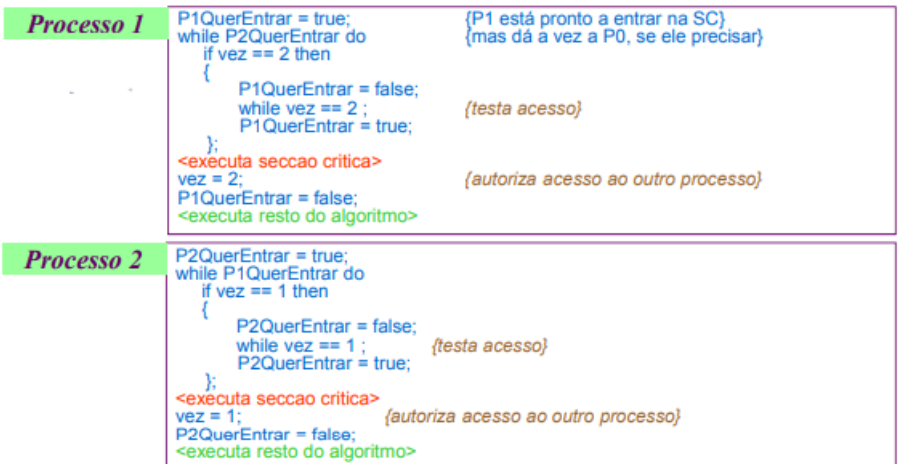

Figura 5.4: Algoritmo seguido pelo modelo.

5.3.1 Criticas ao Algoritmo de Dekker

O algoritmo de Dekker, apesar de ser uma solução clássica para o problema de *Mutual Exclusion* entre dois processos, não está livre de críticas. Algumas dessas críticas estão relacionadas à sua ineficiência, complexidade e limitações em sistemas mais modernos. Estas são as seguintes:

- Ineficiência de desempenho: O algoritmo de Dekker utiliza um esquema de rodízio, onde os processos alternam entre as fases "entrar"e "esperar". Isso resulta em um *looping* de espera, mesmo quando um processo não tem interesse em aceder à secção crítica. Esse *looping* de espera consome recursos de processamento, levando a um desperdício de tempo e energia.

- Complexidade de implementação: O algoritmo de Dekker pode ser considerado complexo de entender e implementar corretamente. O uso de variáveis compartilhadas e a necessidade de sincronização adequada entre os processos podem introduzir problemas de concorrência difíceis de serem identificados e corrigidos, como *deadlocks* e *race conditions*.
- Limitação em sistemas com mais processos: O algoritmo de Dekker foi projetado para lidar com *Mutual Exclusion* entre dois processos. Se houver a necessidade de mais processos o algoritmo não é adequado e é necessário recorrer a soluções mais complexas.
- Falta de justiça no acesso à secção crítica: Dependendo da ordem de execução dos processos e do valor inicial das variáveis compartilhada um processo pode ficar bloqueado indefinidamente, sem nunca conseguir acesso à secção crítica.

5.3.2 Modelar em *Uppaal*

Para modelar este algoritmo em *Uppaal* foram a uma única *template* adicionadas 8 Localizações, uma com o nome de "remainder"(também representa o inicio), 6 delas com o nome lX (representam Localizações intermédias do algoritmo), onde 1<=X<=6, e por fim uma com o nome "critical_section"(representa a secção critica). Existem onze *Edges* que consistem no seguinte:

- remainder->l1, com uma *Updates*/Atualizações "flag[pid]=true"que corresponde á linha do algoritmo "P1QuerEntrar = true;", representado a inteção do processo 1 ou 2 querer entrar na secção critica.
- l1->critical_section, com uma *Guards* "flag[1-pid] == false"que verifica se o processo oposto não tem intenção de entrar na secção critica.
- critical_section->l2, com uma *Updates*/Atualizações "turn=1-pid"que atualiza a variavel do turno para o valor do procesos oposto, corresponde a linha do algoritmo "vez = 2;"ou "vez=1"; dependendo do processo que estava na secção critica.
- l2->remainder, com uma *Updates*/Atualizações "flag[pid]=false"que representa a linha do algoritmo "P1QuerEntrar = false;"ou "P2QuerEntrar = false;"dependendo do processo que estava na secção critica, atualizando a inteção do processo correspondente entrar na secção critica.
- l1->l3, com uma guard "flag[1-pid] == true"que representa a verificação da intenção do processo oposto de entrar na secção critica.

- l3->1, com uma guard "turn==pid"que representa a verificação de quem é a vez de entrar no secção critica.

- l3->l4, com uma guard "turn==1-pid"que representa a verificação de quem é a vez de entrar no secção critica.

- l4->l5, com uma *Updates*/Atualizações "flag[pid]=false"que atualiza a intenção do processo entrar na secção critica para falso, representa a linha do algoritmo "P1QuerEntrar = false;"ou "P2QuerEntrar = false;"dependendo do processo.

- l5->l5, com uma *Guards* "turn==1-pid"que verifica a vez de quem pode entrar na secção critica, mantendo o processo que não pode na Localizações l5, representa a linha de algoritmo "while vez == 1 ;"ou "while vez == 2 ;".

- l5->l6, com uma *Guards* "turn==pid"que verifica a vez de quem pode entrar na secção critica, permitindo avançar para l6 caso seja a vez do processo.

- l6->l1, com uma *Updates*/Atualizações "flag[pid]=true", que vai atualizar a intenção do processo entrar na secção critica para verdade, representa a linha de algoritmo "P1QuerEntrar = true;"ou "P2QuerEntrar = true;", dependendo do processo.

Em suma, cada *Edges* representa uma linha do algoritmo. Foi adicionado à *template* o parâmetro "const int pid", para identificar o processo, a variável "int turn:=1", que representa a vez do processo, e a variável "bool flag[2]"que representa a intenção do processo entrar ou não na secção critica. Isto gera o seguinte modelo no *Uppaal*:

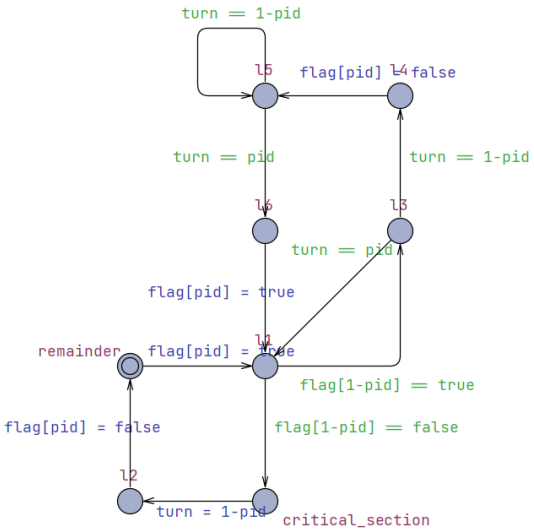

Figura 5.5: Modelo em *Uppaal* do algoritmo de Dekker.

5.3.3 Verificações

O sistema foi verificado usando o verificador do *Uppaal* para garantir que este corre como o algoritmo é suposto.

5.3.3.1 *Safety*

Não aplicável.

5.3.3.2 *Mutual Exclusion*

- A[](P0.critical_section imply not P1.critical_section), verifica que se o processo 1 estiver na secção critica o processo 2 não está lá.

- A[] (P1.critical_section imply not P0.critical_section), verifica que se o processo 2 estiver na secção critica o processo 1 não está lá.

5.3.3.3 *Deadlock*

- A[] not deadlock

5.3.3.4 *Utility*

- E<> (P1.remainder imply P1.critical_section), verifica que eventualmente se o processo 2 estiver no remainder ele vai aceder à secção critica.
- E<> (P0.remainder imply P0.critical_section), verifica que eventualmente se o processo 1 estiver no remainder ele vai aceder à secção critica.

5.3.3.5 *Reachability*

- E<>(P1.critical_section), verifica que eventualmente o processo 1 estará na secção critica.
- E<>(P0.critical_section), verifica que eventualmente o processo 2 estará na secção critica.

Pode ser visto o sucesso das verificações na imagem seguinte:

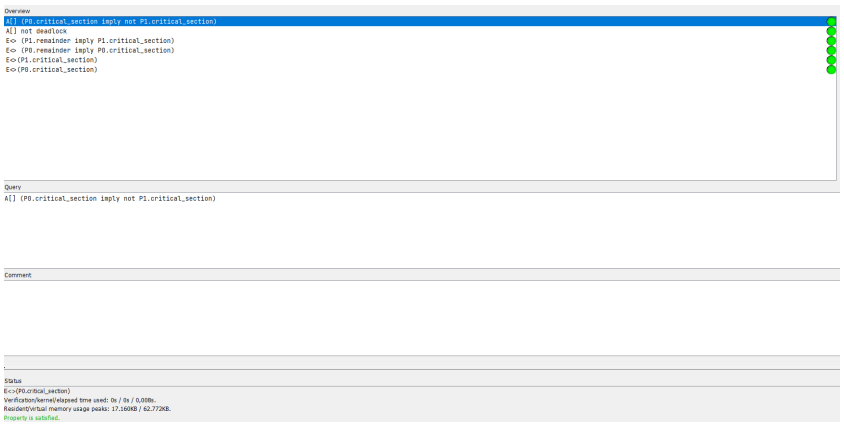

Figura 5.6: Captura de ecrã do verificador do *Uppaal* que mostra o sucesso das propriedades indicadas em cima.

5.4 Algoritmo de Peterson

O algoritmo de Peterson é um algoritmo de sincronização usado na ciência da computação para lidar com o problema da secção crítica na programação concorrente. O problema da secção crítica surge quando vários processos precisam aceder um recurso compartilhado ou uma secção de código que não deve ser executada simultaneamente por mais de um processo. Este foi batizado em homenagem a Gary L. Peterson que o introduziu em 1981

O algoritmo pressupõe a existência de dois processos que tentam aceder a secção crítica. Utiliza duas *flags* booleanas, uma para cada processo, para indicar se um processo está interessado em entrar na secção crítica. Além disso, há uma variável para indicar de quem é a vez de entrar na secção crítica.

É um algoritmo básico e fácil de modelar no *Uppaal* e foi uma das minhas introduções à ferramenta para perceber como trabalhar com a mesma. O algoritmo é mostrado na imagem seguinte:

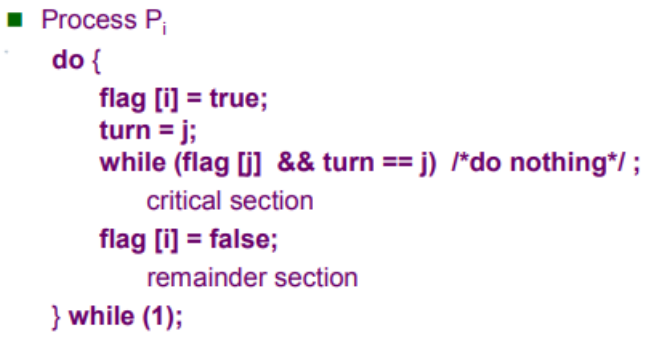

Figura 5.7: Código seguido pelo algoritmo.

5.4.1 Criticas ao Algoritmo de Peterson

O algoritmo de Peterson, apesar de ser uma solução clássica para o problema da seção crítica em programação concorrente, não está isento de críticas e limitações.

- No algoritmo, os processos que não possuem a vez de entrar na secção crítica permanecem em um *loop* de espera, verificando constantemente se é a sua vez. Isso resulta num desperdício de recursos de processamento, uma vez que os processos ficam ocupados à espera, em vez de libertar o processador para outras tarefas. Isto é chamado uma Espera Ativa ou *Busy Waiting*

- Assume a presença de apenas dois processos. Embora funcione bem nesse cenário, ele não escala para um número maior de processos. À medida que o número de processos aumenta, a complexidade do algoritmo cresce exponencialmente, tornando-o menos eficiente e difícil de manter.

- Requer a cooperação dos processos para funcionar corretamente. Se um processo não seguir corretamente o protocolo do algoritmo, ele pode

interferir no acesso à secção crítica de outros processos, resultando em condições de corrida ou bloqueios permanentes.

- Não lida com outras questões importantes da programação concorrente, como a possibilidade de um processo ser interrompido durante a execução da secção crítica. Também não aborda mecanismos de prioridade entre os processos, o que pode levar a problemas de inversão de prioridade.

Em suma, embora o algoritmo de Peterson seja uma solução clássica e útil para o problema da secção crítica, ele apresenta limitações significativas.

5.4.2 Modelar em *Uppaal*

Para modelar este algoritmo em *Uppaal* foram a uma única *Template* adicionadas 4 Localizações, uma com o nome de "idle"(representa o inicio do sistema), outra com o nome "want"(representa que o processo quer aceder à secção critica), outra com o nome "wait"(representa o estado onde o processo espera que a secção critica fique disponível) e outra com o nome "cs"(que representa a secção critica). Existem 5 *Edges* que consistem no seguinte:

- idle->want, com um *Updates*/Atualizações "req_self:=1"(vai atualizar o valor da variável req_self para 1, esta variavél representa o interesse do processo atual em entrar na secção crítica.)
- want->wait, com uma *Updates*/Atualizações "turn:=(me==1?2:1)"(vai atualizar o o valor da variável "turn", que representa a vez do processo a acessar a secção critica, para 2 se o valor atual for da variável "me", que representa o processo atual, for igual a 1 e vice-versa.
- wait->cs, com uma *Guards* "turn==me"(vai verificar se é vez do processo aceder à secção critica).
- wait->cs, com uma *Guards* "req_other==0"(verifica se a variável "req_other"é igual a zero. A variável "req_other"é usada para indicar se o outro processo está interessado em entrar na secção crítica, se "req_other"for igual a zero, isso significa que o outro processo quer entrar na secção crítica no momento).
- cs->idle, com uma *Updates*/Atualizações "req_self:=0"(atualiza a variável "req_self"para 0 (indicando que o processo atual não está mais interessado em entrar na secção crítica, isso permite que o outro processo, caso esteja interessado, possa entrar na secção crítica na próxima iteração do algoritmo).

Os seguintes parâmetros foram introduzidos na *template* para garantir que estas *Edges* ocorrem corretamente: "const int[1,2] me, int[0,1] &req_self, int[0,1] &req_other". Além disso foram definidas as seguintes declarações globais: "int[0,1] req1,req2; int[1,2] turn;"e os templates instanciados assim:

- P1 = mutex(1, req1, req2);
- P2 = mutex(2, req2, req1);

Comparando com o código original 5.7 as duas variáveis "req"representam o *array* de dois *booleans* "flag"com valores de zero (falso) e 1(verdado) por exemplo: "flag[0]==req1"e "flag[1]==req2"e a variavél "turn"representa a variavel "turn"na mesma.

Isto tudo resulta no seguinte modelo para este algoritmo:

O algoritmo é o seguinte:

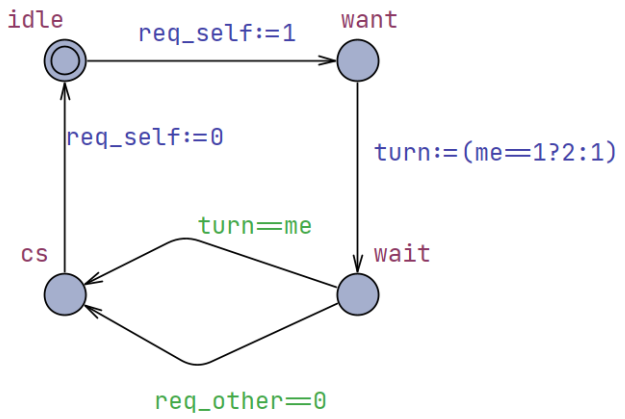

Figura 5.8: Modelo em *Uppaal* do Algoritmo de Petersons.

5.4.3 Verificações

O sistema foi verificado usando o verificador do *Uppaal* para garantir que este corre como o algoritmo é suposto.

5.4.3.1 *Mutual Exclusion*

- A[]!(P1.cs && P2.cs), garante que ambos os processos não estão na secção critica ao mesmo tempo.

5.4.3.2 *Deadlock*

- A[]!deadlock, verifica que não existem deadlocks no modelo.

5.4.3.3 *Utility*

- E<>(P1.idle imply P1.cs), verifica que ao estar no estado inicial, o processo 1 eventualmente entra na secção critica.
- E<>(P2.idle imply P2.cs), verifica que ao estar no estado inicial, o processo 2 eventualmente entra na secção critica.

5.4.3.4 *Reachability*

- E<> P1.cs, verifica que eventualmente o processo 1 entra na secção critica.
- E<> P2.cs, verifica que eventualmente o processo 2 entra na secção critica.

Pode ser visto o sucesso das verificações na imagem seguinte:

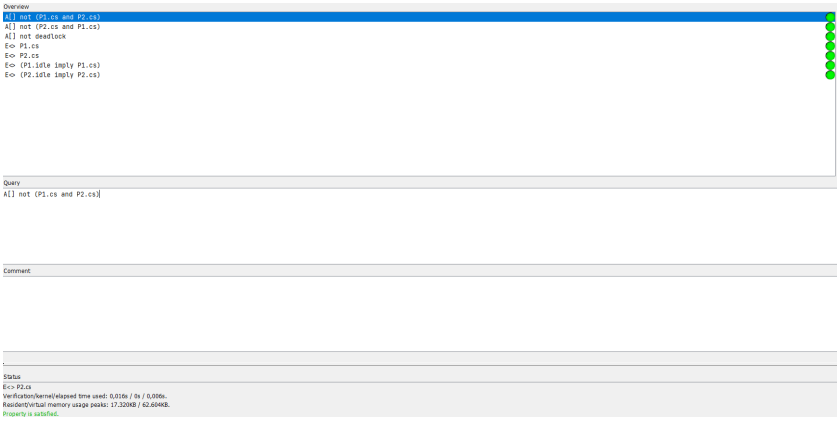

Figura 5.9: Captura de ecrã do verificador do *Uppaal* que mostra o sucesso das propriedades indicadas em cima.

5.5 Algoritmo de Peterson versão errada

Além do algoritmo de Peterson modelado em cima foi também modelada uma versão alternativa que não garante *Mutual Exclusion* e por consequência é uma versão errada do problema.

Ao contrário do algoritmo original de Peterson, que garantia a *Mutual Exclusion* entre processos concorrentes, esta versão modificada não possui um mecanismo efetivo para evitar a execução simultânea das secções críticas. A condição de saída do *loop while* não leva em consideração a existência de um processo em sua secção crítica, logo múltiplos processos podem aceder a secção crítica ao mesmo tempo, resultando em uma violação da *Mutual Exclusion*.

Segue o seguinte algoritmo modificado:

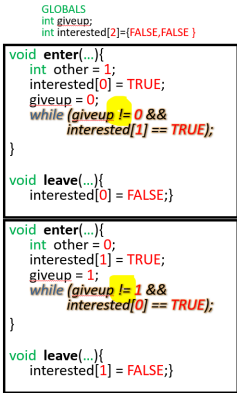

Figura 5.10: Algoritmo seguido para modelar o Algoritmo de Peterson Errado.

5.5.1 Modelar em *Uppaal*

Para modelar este algoritmo em *Uppaal* foram a uma *template* adicionadas quatro Localizações, sendo elas o "start"(representa o estado inicial), o idle e want (representam Localizações intermédias) e a "cs"(representa a secção critica). Estas Localizações foram unidas por *Edges* que respeitam o seguinte:

- start->idle, com a *Updates*/Atualizações "interested[pid]=true"que atualiza a intenção do processo entrar na secção critica como verdade.

- idle->want, com a *Updates*/Atualizações "giveup=pid"que representa a linha do código "giveup=0;"ou "giveup=1;".

- want->cs, com as *Guards* "giveup!=pid && interested[1-pid]==true"que verificam se processo reúne os requisitos para entrar na secção critica, corresponde à linha de código do algoritmo igual. Tem também a *Updates*/Atualizações "giveup=pid"para permitir ter os dois processos na secção critica.

- cs->idle, com a *Updates*/Atualizações "interested[pid]", que reseta a intenção do processo entrar na secção critica.

Foi também adicionado o parâmetro "const int pid"para identificar os processos e definidas as variáveis "int giveup; int interested[2] ={false,false}". Isto cria o seguinte modelo:

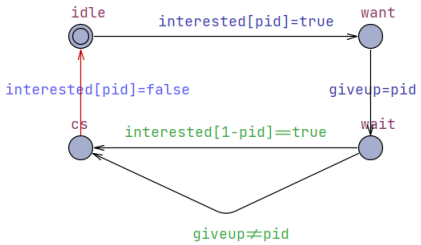

Figura 5.11: Modelo do Algoritmo de Peterson errado em *Uppaal*.

5.5.2 Verificações

O sistema foi verificado usando o verificador do *Uppaal* para garantir que este corre como o algoritmo é suposto tendo em conta que a intenção era para mostrar que esta nova versão de Petersons de facto **não garante** a *Mutual Exclusion*.

- A[] !deadlock, verifica que não há deadlock.

- E<> (Process.cs && Process2.cs), verifica que não existe *Mutual Exclusion* uma vez que existe um estado onde os processos 1 e 2 vão estar ambos na secção critica ao mesmo tempo.

A veracidade destas propriedades pode ser verificada na imagem seguinte:

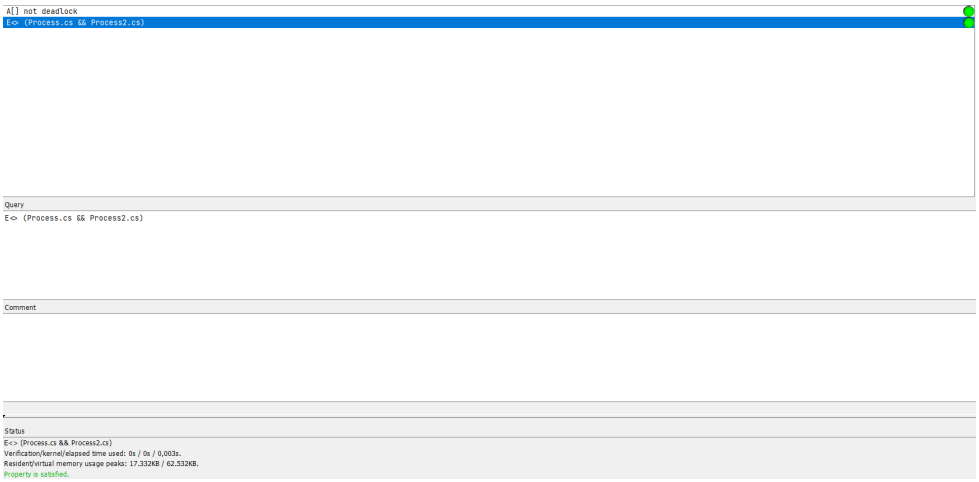

Figura 5.12: Captura de ecrã que verifica a veracidade das propriedades acima.

5.6 Algoritmo da Confeitaria de Lamport

O Algoritmo da Confeitaria de Lamport (*Lamport's Bakery Algorithm*) é um algoritmo clássico utilizado para garantir a *Mutual Exclusion* entre N processos concorrentes. Cada processo recebe uma senha de entrada na secção crítica e utiliza o algoritmo para determinar qual processo terá acesso à secção crítica em determinado momento.

O funcionamento do algoritmo de Lamport é baseado nos seguintes passos:

Atribuição de senhas: A cada processo é atribuída uma senha de entrada na secção crítica. Antes de aceder a secção crítica, o processo executa a função "máximo"(max), que determina uma senha com numero superior a todas as outras. É possível que vários processos tenham a mesma senha, isto não pode ser evitado sem antes resolver o problema de *Mutual Exclusion* que é o objetivo do próprio algoritmo.

Comparação de senhas: Antes de entrar na secção crítica, o processo verifica se a sua senha é menor do que a de todos os outros processos em execução. Essa comparação é realizada através da função "comparar"(compare). Caso a senha do processo seja a menor de todas, ele pode prosseguir para a próxima etapa.

Ciclo de espera: Após ter atribuída a senha e verificado que é a menor, o processo entra em um ciclo de espera, onde ele aguarda até que seja o seu número de ordem a ser atendido. Esse ciclo é importante para garantir a or-

dem correta de acesso à secção crítica, evitando a ocorrência de condições de corrida.

Acesso à secção crítica: Quando o número da senha do processo for inferior ao de todos os outros, o processo pode entrar na secção crítica. Nesse momento, ele pode executar as operações desejadas com segurança, sabendo que não haverá interferência de outros processos.

O algoritmo de Lamport é uma solução eficaz para a *Mutual Exclusion* em sistemas com N processos. Ele garante que apenas um processo entre na secção crítica de cada vez, seguindo uma ordem determinada pelas senhas atribuídas. No entanto, é importante estar ciente de suas limitações

Segue o seguinte algoritmo:

```
Shared senha : array[0..N-1] of integer;
       escolha : array[0..N-1] of boolean;

function compare(a, b:integer): boolean;
begin
   if (senha[a]<senha[b])
         or (senha[a]==senha[b] and a<b)
         //desempatar pelo nome/id
     then return true;  else return false;
end;

function max: integer;
integer maximo:=0, i;
begin
   for i:=1 to N do
         if ( senha[i]>maximo )
              then maximo := senha[i];
   return maximo;
end;
```

```
procedure PROC(id:integer);
integer j;
begin
   while true do  {protocolo tem que fuuncionar em ciclo }
   begin
      escolha[id]:=true;        {atribuicao da senha}
      senha[id]:=max + 1;
      escolha[id]:=false;
      for j:=0 to N-1 do        {espera ate senha menor}
      begin
         while escolha[j] do;
         while (senha[j]<>0) and compare(j,id) do;
      end;
      <executa seccao critica>
      senha[id]:=0;
      <executa resto do algoritmo>
   end
end;
```

Figura 5.13: Algoritmo seguido para a modelação do algoritmo de Lamport.

5.6.1 Criticas ao Algoritmo de Lamport

Mesmo sendo uma solução eficaz em muitos casos, o algoritmo também recebe críticas que devem ser consideradas:

- Complexidade e custo computacional: O algoritmo de *Bakery's* Lamport possui uma complexidade considerável, principalmente quando o número de processos aumenta. Cada processo deve executar a função "max"para determinar a sua senha, o que pode resultar em um alto custo computacional em sistemas com muitos processos. Isso pode afetar o desempenho geral do sistema.

- Falta de escalabilidade: À medida que o número de processos aumenta, a escalabilidade do algoritmo de *Bakery's* Lamport torna-se um desafio. À medida que mais processos competem pelo acesso à secção crítica, o tempo de espera aumenta consideravelmente. Isso pode levar a um problema conhecido como "*starvation*", onde alguns processos podem

ter dificuldade em aceder a secção crítica devido ao tempo de espera excessivo.

- Limitações na distribuição: O algoritmo de *Bakery's* Lamport foi projetado principalmente para sistemas centralizados, onde a comunicação entre os processos é direta. Em ambientes distribuídos, onde os processos podem estar em diferentes nós ou máquinas, o algoritmo pode tornar-se menos eficiente ou até mesmo inviável devido às complexidades adicionais introduzidas pela comunicação em rede.

Em conclusão, embora o algoritmo de Bakery's Lamport seja amplamente reconhecido e utilizado para garantir a *Mutual Exclusion* entre processos, ele não está isento de críticas.

5.6.2 Comparação com o Algoritmo de Eisenberg e McGuire

Durante o desenvolvimentos deste projeto também implementamos um modelo para o algoritmo de Eisenberg e McGuire. O modelo foi obtido do [17] e verificamos as propriedades que foram descritas neste artigo. O modelo implementado está disponível no Github do projeto.

Ao comparar o algoritmo de *Bakery's* Lamport e o algoritmo de Eisenberg e McGuire, que são utilizados para resolver o problema de *Mutual Exclusion* entre N processos, podemos identificar diferenças significativas.

O algoritmo de Lamport é uma solução clássica que atribui senhas aos processos para determinar a ordem de acesso à secção crítica. Ele utiliza um esquema de rodízio, onde cada processo recebe uma senha superior às demais e aguarda a sua vez para aceder a secção crítica. Embora seja eficiente em sistemas com um número limitado de processos, pode enfrentar desafios de escalabilidade à medida que o número de processos aumenta.

Por outro lado, o algoritmo de Eisenberg e McGuire é uma melhoria em relação ao algoritmo de Bakery's Lamport. Ele introduz uma fase adicional chamada "proposta", onde os processos indicam a sua intenção de aceder a secção crítica e aguardam até que sua proposta seja aceita. Essa modificação torna o algoritmo mais eficiente em termos de tempo de espera e mais adequado para sistemas distribuídos.

Em termos práticos, o algoritmo de Eisenberg e McGuire reduz o tempo de espera, evitando que processos aguardem desnecessariamente quando outros processos não têm interesse em aceder a secção crítica. Isso torna o algoritmo mais eficiente em ambientes distribuídos, onde a comunicação entre processos pode introduzir atrasos e impactar o desempenho geral.

5.6.3 Problemas encontrados com a modelação de Lamport em *Uppaal*

O maior problema encontrado ao modelar o algoritmo foi o facto das senhas serem do tipo "*unbounded integer*"(inteiro ilimitado). Isto fazia com que o *Uppaal* tivesse problemas em provar propriedades de tais sistemas devido ao problema de "*state explosion*"(explosão do numero de estados com os quais o *Uppaal* tem que lidar).

A solução para isto foi definir o numero máximo da senha como 100 e a partir deste número começar a contar do zero novamente tendo em atenção as situações em que por exemplo a senha 99 tem que ser considerada como menor que a senha 12 visto que a senha 12 foi vitima de um *reset* de senhas e a senha 99 não. Para conseguir isto a função "compare"foi modificada da inicialmente descrita para a seguinte e

```
bool compare(int a, int b) {
  if (senha[a] <= 100 &&  senha[b] <= 100 && senha[a]<senha[b]) {
    return true;
  }

  if (senha[a] == senha[b]) {
    if (a < b)
      return true;
    else
      return false;
  }

  if(senha[b] == 0) return true;

  if (senha[a] > 80 && senha[b]<20) return true;

  return false;
}
```

Excerto de Código 5.1: Função compare que compara as senhas e verifica qual é a menor para dar prioriodade a esta. Esta função tem em atenção o reset de senhas quando estas são maiores que cem.

Esta função considera problemas como o indicado em cima em especifico na linha "if (senha[a] > 80 && senha[b]<20) return true;", sendo o código auto-explicativo.

Foi também modificada a função "max"da inicialmente prevista de forma a permitir a resolução do problema acima para a seguinte:

```
  int max() {
    int v = senhamax;
    senhamax = (senhamax+1) % 100;
```

```
    return v;
}
```

Excerto de Código 5.2: Função max que calcula qual é a senha maior dos processos.

E também temos em conta que introduzimos um *"clock"* no estado inicial fazendo com que um processo tivesse que sair deste estado dentro dum certo tempo (pusemos 6) - feita assim as senhas não podem diferir entre elas com um valor maior que seis (quer dizer que um processo pode entrar na secção critica um máximo de 6 vezes, adquirindo e incrementado assim a senha, antes dum outro processo também sair deste estado para entrar na secção critica, além de outras condições que serão expostas abaixo).

5.6.4 Modelar em *Uppaal*

Para modelar este algoritmo em *Uppaal* foram a uma única *template*. adicionadas 9 Localizações, uma com o nome de "start"(representa o inicio), 7 delas com o nome lX (representam Localizações intermédias do algoritmo), onde 1<=X<=7, e por fim uma com o nome "cs"(representa a secção critica). Existem doze *Edges* que consistem no seguinte:

- start->l1, com a *Guards* "x[id]>=2"que verifica se o clock é superior ou igual a dois e com a *Updates*/Atualizações "escolha[id]=true"que atualiza a variável escolha[id] para verdade indicando que aquele processo já escolheu.
- l1->l2, com a *Updates*/Atualizações "senha[id]=max()+1"que vai atribuir uma senha superior à máxima anterior ao processo.
- l2->l3, com a *Updates*/Atualizações "escolha[id]=false"que atualiza a variável escolha para falso indicando que que já não foi escolhido.
- l3->l4, com a *Updates*/Atualizações "j=0"que inicia o loop do algoritmo.
- l4->l4, com a *Guards* "escolha[id]==true"que verifica que se a variável escolha para aquele processo for verdade mantém o mesmo naquela Localizações.
- l4->l5, com a *Guards* "escolha[j]==false"que permite o processo avançar no loop caso o valor desta variável seja falsa.
- l5->l5, com a *Guards* "senha[j]!=0 && compare(j,id)"que mantém o processo nesta Localizações enquanto estas condições forem verdadeiras.

- l5->l6, com a *Guards* "senha[j]==0 || compare(j,id) ==false"que permite o avanço para a Localizações l6 caso estas sejam verdadeiras, tem também uma *Updates*/Atualizações "j=j+1"que incrementa o valor de j.
- l6->l4, com uma *Guards* "j<N"que verifica a condição final do loop.
- l6->cs, com uma *Guards* "j==N"e uma *Updates*/Atualizações "y[id]=0"que reseta o valor do clock y.
- cs->l7, com a *Updates*/Atualizações "senha[id]=0"que reseta o valor da senha do processo.
- l7->start, com a *Updates*/Atualizações "x[id]=0"que reseta o valor do clock x.

Foi declarado o parâmetro "const int[0, N-1] id"para identificar o processo, além disso foram definidas as variáveis "const int N=4;"(representa o numero de processos), "int senha[N]=0,0,0,0;"(representa a senha do processo que é inicializada em zero), "bool escolha[N]=false,false,false,false (representa se já foi escolhido ou não);"e "clock x[N],y[N];".

Além disso foram criadas as funções indicadas em cima.

Isto tudo origina o seguinte modelo em *Uppaal*:

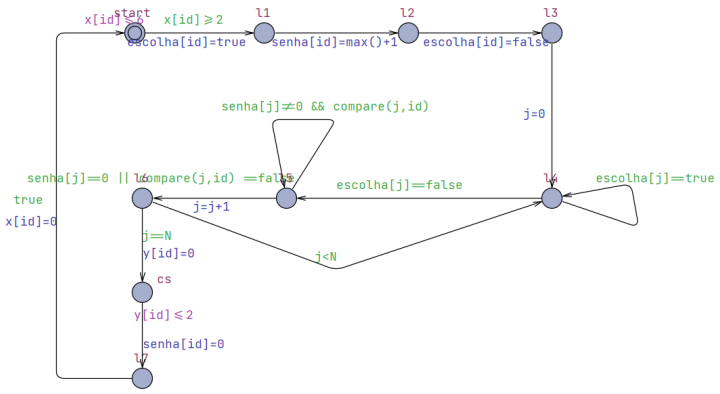

Figura 5.14: Modelo do Algoritmo de Lamport em *Uppaal*.

5.6.5 Verificações

O sistema foi verificado usando o verificador do *Uppaal* para garantir que este corre como o algoritmo é suposto.

5.6.5.1 *Safety*

Não aplicável.

5.6.5.2 *Mutual Exclusion*

- A[] (forall(i:pid) Process(i).cs imply (forall(j:pid) j!=i imply !Process(j).cs)), garante que se um processo está na secção crítica, nenhum outro processo pode estar executando sua secção crítica ao mesmo tempo.

- A[] (Process(0).cs imply not Process(1).cs), verifica que se o processo 1 estiver na secção critica o processo 2 não está lá.

- A[] (Process(1).cs imply not Process(0).cs)), verifica que se o processo 2 estiver na secção critica o processo 1 não está lá.

5.6.5.3 *Deadlock*

- A[] !deadlock, garante que não existe deadlock no sistema.

5.6.5.4 *Utility*

- E<> (P1.start imply P1.cs), verifica que eventualmente se o processo 2 estiver no start ele vai aceder à secção critica.

- E<> (P0.start imply P0.cs), verifica que eventualmente se o processo 1 estiver no start ele vai aceder à secção critica.

5.6.5.5 *Reachability*

- E<>(P1.cs), verifica que eventualmente o processo 1 estará na secção critica.

- E<>(P0.cs), verifica que eventualmente o processo 2 estará na secção critica.

Pode ser visto o sucesso das verificações na imagem seguinte:

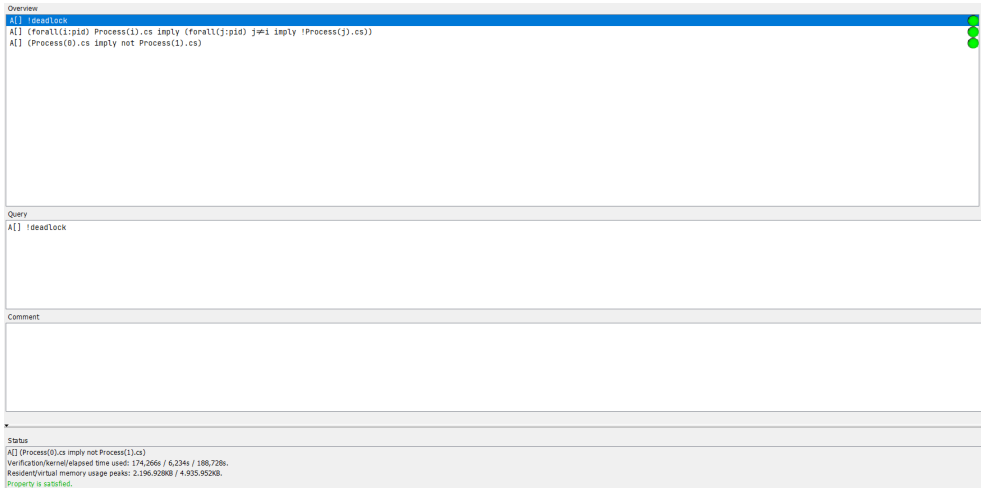

Figura 5.15: Captura de ecrã do verificador do *Uppaal* que mostra o sucesso das propriedades indicadas em cima.

5.7 Cenários de *Deadlock*

Para terminar esta secção abordamos agora dois problemas relacionados com *Deadlock* que habitualmente são discutidos no contexto dum curso de sistemas operativos, por exemplo no livro [16]. Neste livro um sistema é modelado usando um grafo que embora possa ser anilasado usando algoritmos de grafo para determinar se existe ou não uma situação de *Deadlock* não permite um analise mais dinâmico como uma ferramenta como *Uppaal* permite. Os recursos nesta situação não são preemptiveis, quer dizer não podem ser retirado do posse dum processo sem que a computação falha. Cada processo utilize um recurso na seguinte maneira, requisitar o recurso, utilização do recurso e libertar o recurso. Os dois cenários modeladas a seguir são exercícios do curso de Sistemas Operativos.

5.7.1 *Deadlock* 1

Abordaremos a modelagem de um cenário hipotético que envolve 3 processos e 4 recursos, com o objetivo de analisar a ocorrência de um impasse, também conhecido como *Deadlock*. Nessa situação, iremos utilizar a ferramenta *Uppaal* para simular e compreender o comportamento do sistema. O sistema consiste nos seguintes processos e atribuições iniciais de recursos:

- O Processo P1 possui o recurso R1 atribuído, mas está aguardando a disponibilidade do recurso R2;

- O Processo P2 possui o recurso R2 atribuído, mas está esperando pela libertação do recurso R3;

- O Processo P3 possui os recursos R3 e R4 atribuídos, mas está à espera do recurso R1.

Essa configuração cria uma interdependência complexa entre os processos, na qual cada um deles está à espera de recursos que estão a ser utilizados por outros processos. Essa situação pode levar a um estado de impasse, no qual nenhum dos processos pode progredir, resultando em uma paralisação do sistema.

Para modelar esse cenário no *Uppaal*, iremos definir os processos como autómatos, especificar as *Edges* e restrições de alocação de recursos, e simular a execução do sistema. Ao observar a simulação, será possível identificar se ocorre um *Deadlock*, ou seja, se os processos ficam bloqueados indefinidamente, incapazes de liberar os recursos que estão sendo aguardados por outros processos.

5.7.1.1 Modelar em *Uppaal*

Para modelar este *Deadlock* em *Uppaal* foi usada uma única *template*. Foram adicionadas dez Localizações, uma com o nome de "Start"(representa o inicio) e 9 que representam o estado dum processo X estar atribuído ao recurso Y (exemplo: p1_r1, significa que o processo 1 está com o recurso 1).

Nota que o enunciado original represente um *snapshot* ou momento em tempo e podemos modelar um sistema mais flexível com *Uppaal*, por exemplo o processo P1 necessita de dois recursos para avançar mas podemos flexibilizar o modelo deixando o processo adquirir R1 ou R2 primeiro.

Com estes reparos existem assim dezassete *Edges* que consistem no seguinte:

- start->p1_r1 , com uma *Guards* "p == 1 && r1 == 0", que verifica se o processo é o 1 e se o recurso 1 está livre, com uma *Updates*/Atualizações "r1:=1"que bloqueia o recurso 1.

- p1_r1->p1_r1_r2, com uma *Guards* "r2==0"e uma *Updates*/Atualizações "r2:=1"que verificam que o recurso 2 está livre e caso este esteja ocupa o mesmo respetivamente.

- p1_r1_r2->start, com uma *Updates*/Atualizações "r1:=0,r2:=0"que liberta os recursos 1 e 2.

- start->p1_r2, com uma *Guards* "p==1 && r2==0"que verifica se o processo é o 1 e se o recurso 2 está livre, tem também uma *Updates*/Atualizações "r2:=1"que marca o recurso 2 como ocupado.

- p1_r2->p1_r1_r2, com uma *Guards* "r1==0"e uma *Updates*/Atualizações "r1:=1"que verificam se o recurso 1 está livre e se estiver ocupa o mesmo.

- start->p2_r2, com uma *Guards* "p == 2 && r2 == 0", que verifica se o processo é o 2 e se o recurso 2 está livre, com uma *Updates*/Atualizações "r2:=1"que bloqueia o recurso 2.

- p2_r2->p2_r2_r3, com uma *Guards* "r3==0"e uma *Updates*/Atualizações "r3:=1"que verificam que o recurso 3 está livre e caso este esteja ocupa o mesmo respetivamente.

- p2_r2_r3->start, com uma *Updates*/Atualizações "r2:=0,r3:=0"que liberta os recursos 2 e 3.

- start->p2_r3, com uma *Guards* "p == 2 && r3 == 0", que verifica se o processo é o 2 e se o recurso 3 está livre, com uma *Updates*/Atualizações "r3:=1"que bloqueia o recurso 3.

- p2_r3->p2_r2_r3, com uma *Guards* "r2==0"e uma *Updates*/Atualizações "r2:=1"que verificam que o recurso 2 está livre e caso este esteja ocupa o mesmo respetivamente.

- start->p3_r1_r4, com uma *Guards* "p == 3 && r1 == 0", que verifica se o processo é o 3 e se o recurso 1 está livre, com uma *Updates*/Atualizações "r1:=1"que bloqueia o recurso 1.

- p3_r1_r4->p3_r3_r4_r1, com uma *Guards* "r3 == 0 && r4 == 1"que verifica se o recurso 2 está livre e se o recurso 4 está ocupado, tem também uma *Updates*/Atualizações "r3:=1"que marca o recurso 3 como ocupado.

- p3_r3_r4_r1->start, com uma *Updates*/Atualizações "r3 := 0,r4 := 0,r1 := 0"que liberta os recurso 1,3 e 4.

- start->p3_r3_r4, com uma *Guards* "p == 3 && r3 == 0"que verifica se o processo é o 3 e se o recurso 3 está livre, tem também uma *Updates*/Atualizações "r3:=1"que marca o recurso 3 como ocupado.

- p3_r3_r4->p3_r3_r4_r1, com uma *Guards* "r1 == 0 && r4 == 1"que verifica se o recurso 1 está livre o recurso 4 está ocupado, tem também uma *Updates*/Atualizações "r1 := 1"que marca o recurso 1 como ocupado.

- p3_r3_r4->p3_r3_r4, com uma *Guards* "r4==0"que verifica se o recurso 4 está livre e uma *Updates*/Atualizações "r4:=1"que marca o mesmo como ocupado.

- p3_r1_r4->p3_r1_r4, com uma *Guards* "r4==0"que verifica se o recurso 4 está livre e uma *Updates*/Atualizações "r4:=1"que marca o mesmo como ocupado.

Foi declarado na *template* o seguinte parâmetro "const int p"que serve para atribuir uma identificação numérica aos processos, além disso foram declaradas as seguintes variáveis "int r1=0;int r2=0;int r3=0;int r4=0;"que significam os recursos e iniciam os valores dos mesmos como zero.

Isto origina o seguinte modelo em *Uppaal*:

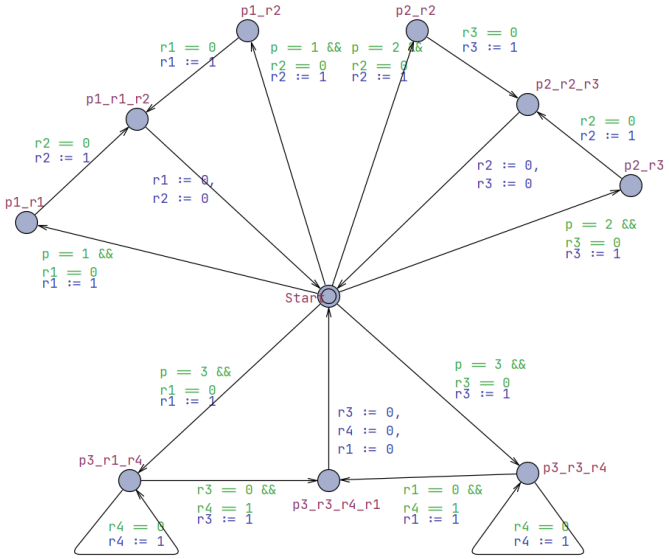

Figura 5.16: Modelo do *Deadlock* 1 em *Uppaal*

5.7.1.2 Verificação do *Deadlock*

Para verificar se o *Deadlock* ocorre no modelo em cima foi utilizado a seguinte expressão lógica:

- E<> deadlock, verifica que eventualmente acontecerá *Deadlock*

Verifica no verificador do *Uppaal* a seguir:

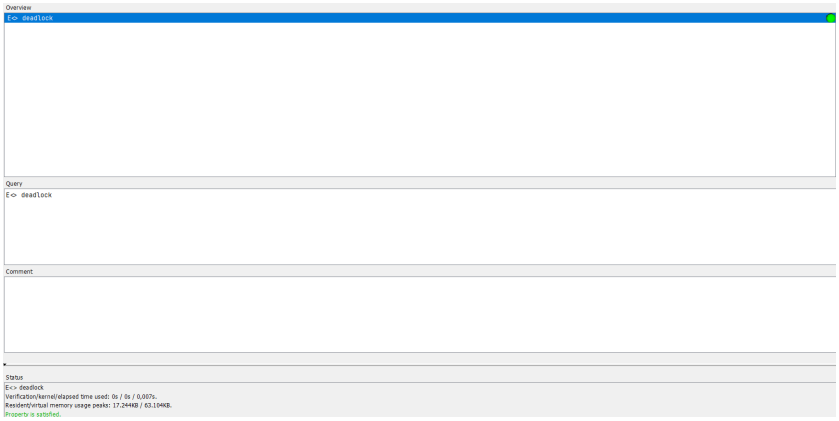

Figura 5.17: Captura de ecrã do verificador do *Uppaal* que mostra o sucesso das propriedades indicadas em cima.

E para verificar se o *deadlock* ocorreu onde era suposto utilizamos a função do *Uppaal* "Diagnostic Trace-> Fastest"que cria a situação de *deadlock* no *Symbolic Simulator* demonstrando que o modelo criado está correto. A seguinte imagem verifica o *Deadlock* correto para o enunciado:

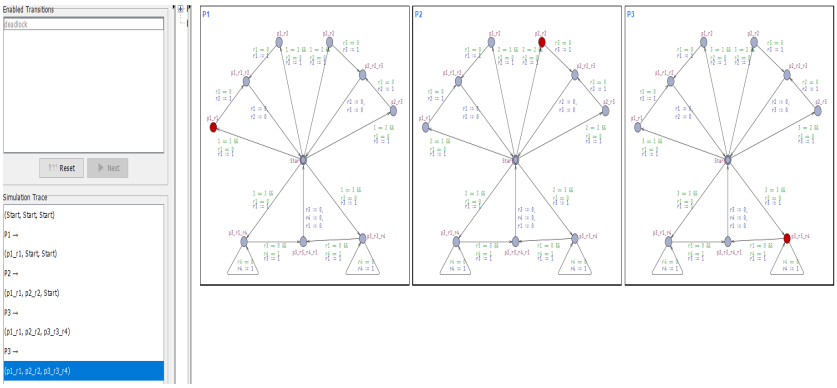

Figura 5.18: Captura de ecrã do simulador do *Uppaal* que prova que há o *Deadlock* suposto.

5.7.2 *Deadlock* 2

Analisaremos uma situação que envolve três processos (P1, P2 e P3) e três recursos (R1, R2 e R3). Nesse caso, os recursos R1 e R3 possuem apenas uma instância cada, enquanto o recurso R2 possui duas instâncias idênticas.

Inicialmente, o processo P1 possui uma das instâncias do recurso R2 atribuída e aguarda a disponibilidade do recurso R1. O recurso R1, por sua vez,

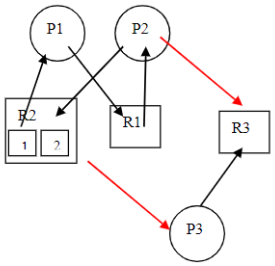

Figura 5.19: Gráfico que permite visualizar o *Deadlock* 2 mais facilmente.

está atribuído ao processo P2. Enquanto isso, o processo P2 espera pela atribuição de uma instância do recurso R2, e o processo P3 está à espera do recurso R3.

Em determinado momento, ocorre uma evolução no sistema. O processo P3 solicita e é atribuída uma instância do recurso R2, enquanto o processo P2 solicita o recurso R3.

Essa nova configuração cria uma situação de possível impasse, na qual cada processo está a espera da libertação de recursos que estão a ser utilizados por outros processos. Essa interdependência pode levar a um estado de *Deadlock*, onde nenhum dos processos pode progredir. O *Deadlock* depende de como o recurso R3 será atribuído. Se for atribuído ao processo P2 haverá *Deadlock*.

Neste subcapítulo, iremos utilizar a ferramenta *Uppaal* para modelar e simular esse cenário que pode ser visualizado da seguinte forma:

5.7.2.1 Modelar em *Uppaal*

Para modelar este *Deadlock* em *Uppaal* foram a uma única *Template* adicionadas 9 Localizações, uma com o nome de "Start"(representa o inicio) e 9 que representam o processo X estar atribuído ao recurso Y (exemplo: p1_r1, significa que o processo 1 está com o recurso 1). Existem doze *Edges* que consistem no seguinte:

- start->p2_r1, com uma *Guards* "p==2 && r1==1"que verifica se o processo é o 2 e se o recurso 1 está livre.

- p2_r1->p2_r1_r2, com uma *Guards* "r2>0"que verifica se existe alguma instância do recurso 2 livre e ocupa a mesma com a *Updates*/Atualizações "r2=r2-1".

- p2_r1_r2->p2_r1_r2_r3, com uma *Guards* "r3==1"que verifica se o recurso 3 está livre e ocupa o mesmo com uma *Updates*/Atualizações "r3:=0".

- p2_r1_r2_r3->start, que liberta os recursos com uma *Updates*/Atualizações "r1:=1 ,r2++, r3=1".

- p2_r1->p2_r1_r3, com uma *Guards* "r3==1"que verifica se o recurso 3 está livre e ocupa o mesmo com a *Updates*/Atualizações "r3:=0".

- p2_r1_r3->p2_r1_r2_r3, com uma *Guards* "r2>0"que verifica se alguma instância do recurso 2 está livre e ocupa a mesma com a *Updates*/Atualizações "r2=r2-1".

- start->p1_r2, com uma *Guards* "p==1&&r2>0"que verifica se o processo é o 1 e se alguma instância do recurso 2 está livre, ocupando o mesmo com a *Updates*/Atualizações "r2=r2-1".

- p1_r2->p1_r2_r1, com uuma *Guards* "r1==1"que verifica se o recurso 1 está livre e ocupa o mesmo com a *Updates*/Atualizações "r1:=0".

- p1_r2_r1->start, que liberta os recursos 1 e 2 com a *Updates*/Atualizações "r2++,r1=1".

- start->p3_r2, com a *Guards* "p==3&&r2>0"que verifica se o processo é o 3 e se alguma instância do recurso 2 está livre, ocupando a mesma com a *Updates*/Atualizações "r2=r2-1".

- p3_r2->p3_r2_r3, com uma *Guards* "r3==1"que verifica se o recurso 3 está livre e ocupa o mesmo com a *Updates*/Atualizações "r3=0".

- p3_r2_r3->start, que liberta o recurso 2 e 3 com a *Updates*/Atualizações "r2++,r3=1".

Na *template* foi também definido o parâmetro "const int p"que serve para numerar os processos e as variáveis "int r1=1,int r2=2, int r3=1"que representam os recursos e inicializam os mesmos. Isto gera o seguinte modelo em *Uppaal*:

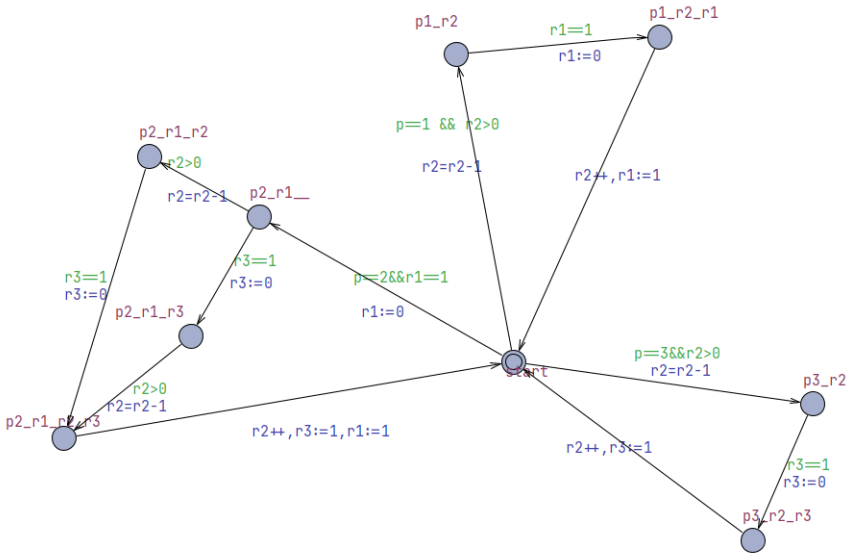

Figura 5.20: Modelo do *Deadlock* 2 em *Uppaal*.

Além disto foi também modelado um modelo protótipo que é global e permite mais *deadlocks* além do proposto inicialmente. Este modelo segue o mesmo raciocínio do inicial mas contem mais Localizações e *Edges* que representam todas as possibilidades dos processos receberem e libertarem recursos. Pode ser observado na seguinte imagem:

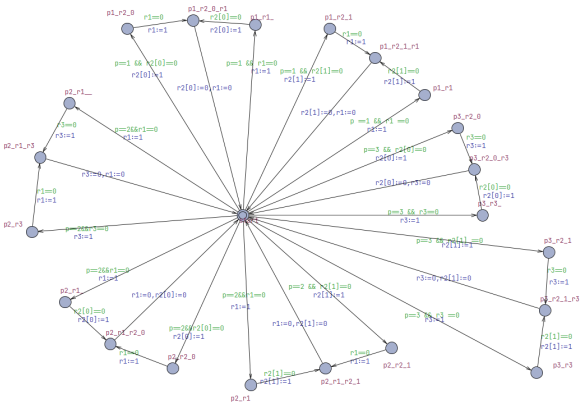

Figura 5.21: Modelo do *Deadlock* protótipo em *Uppaal*.

5.7.2.2 Verificar o Deadlock

Para verificar se o deadlock ocorre nos parâmetros enunciados em cima foi utilizado a seguinte expressão lógica:

- E<> deadlock, verifica que eventualmente acontecerá deadlock

Verifica no verificador do *Uppaal* a seguir:

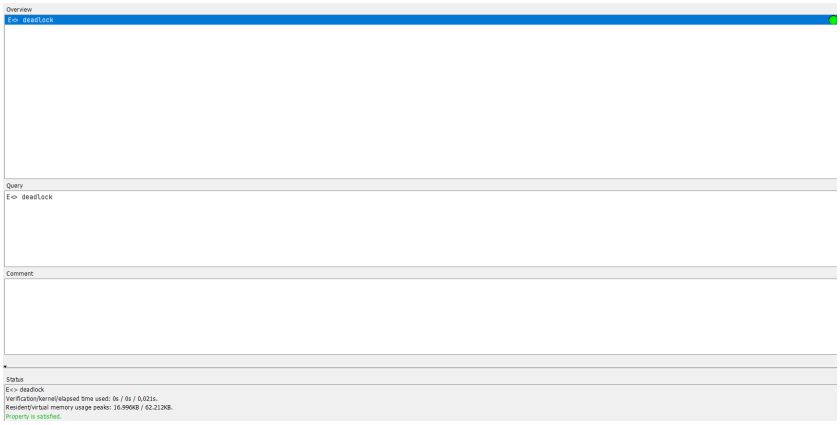

Figura 5.22: Captura de ecrã do verificador do *Uppaal* que mostra o sucesso das propriedades indicadas em cima.

E para verificar se o *deadlock* ocorreu onde era suposto utilizamos a função do *Uppaal* "*Diagnostic Trace-> Fastest*"que cria a situação de *deadlock* no "*Symbolic Simulator*"demonstrando que o modelo criado está correto. A seguinte imagem verifica o *deadlock* correto para o enunciado:

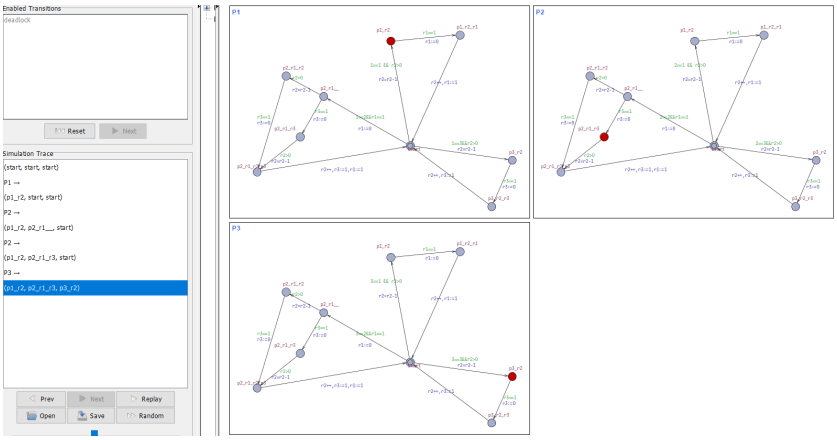

Figura 5.23: Captura de ecrã do simulador do *Uppaal* que prova que há o *Deadlock* suposto.

5.8 Conclusões

Desta forma é concluído este capítulo que demonstrava a nossa implementação de cada algoritmo proposto em *Uppaal*. Foi explicado o que faz cada algoritmo, como este foi interpretado e como foram feitas as decisões para tornarem o algoritmo ser modelado usando a ferramenta *Uppaal*. Foram também mostradas e explicadas as verificações feitas para as propriedades especificadas para cada algoritmo.

Capítulo

6

Conclusões e Trabalho Futuro

6.1 Conclusões Principais

Após utilizar a ferramenta *Uppaal* para modelar algoritmos temporais e de exclusão mútua, aprendi bastante e gostei muito de utilizar essa ferramenta. É possível concluir que o *Uppaal* oferece uma abordagem eficiente e confiável para a verificação e validação destes tipos de algoritmos.

Ao utilizar o *Uppaal*, pude descrever os algoritmos de exclusão mútua de forma clara e concisa, especificando estados, transições e restrições temporais. Essa representação visual facilita a compreensão do comportamento do sistema e a deteção de possíveis problemas ou conflitos.

Além disso, o *Uppaal* oferece recursos avançados de verificação, permitindo analisar propriedades importantes, como a ausência de *deadlocks* e violações de exclusão mútua. Essa capacidade de verificação formal ajuda a garantir que os algoritmos estejam corretos e livre de erros, proporcionando maior confiabilidade e segurança ao sistema.

Durante o processo de modelação, pude identificar e corrigir eventuais falhas e incoerências no algoritmo, evitando assim problemas futuros na implementação e execução do sistema. A capacidade de simulação oferecida pelo *Uppaal* também foi muito útil para validar o comportamento do algoritmo em diferentes cenários e condições.

6.2 Trabalho Futuro

Após a abordagem dos algoritmos de exclusão mútua utilizando o *Uppaal*, existem outros aspetos interessantes que poderiam ser explorados no projeto. Alguns tópicos relevantes incluem:

Integração com ambientes reais: É interessante considerar a integração dos algoritmos modelados em *Uppaal* com um ambiente de execução real. Isso envolve a implementação dos algoritmos numa linguagem de programação específica e sua interação com outros componentes do sistema. A validação experimental dessa integração pode fornecer *insights* valiosos sobre o desempenho e a *Correctness* dos algoritmos em um contexto real.

Escalonamento em sistemas distribuídos: Caso o projeto envolva sistemas distribuídos, pode-se explorar a modelação e a verificação de algoritmos de exclusão mútua adaptados a essa realidade.

Ao abordar esses tópicos adicionais, seria possível elevar o projeto a um nível mais complexo.

Bibliografia

[1] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education, 2014.

[2] L. Chen and P. Hao. Static and dynamic analysis of software verification. In *2015 12th Web Information System and Application Conference (WISA)*, pages 222–226. IEEE, 2015.

[3] J. A. McDermid. *Software verification and validation*. Springer Science e Business Media, 2004.

[4] Dongsun Zhang and Sukyoung Ryu. Dynamic software verification: Challenges and prospects. In *2011 11th International Conference on Quality Software (QSIC)*, pages 315–316. IEEE, 2011.

[5] Manfred Broy and Gunnar Stålmarck. *Software Systems and Software-Integrated Systems*. Springer Science e Business Media, 2013.

[6] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2016.

[7] Nabendu Chaki, Rituparna Chaki, and Saikat Ray. Software verification and validation: problems and their impact on software quality. In *2011 International Conference on Communication Systems and Network Technologies*, pages 398–403. IEEE, 2011.

[8] Nathaniel Ayewah, William Pugh, David Morgenthaler, John Penix, and Yuqiang Zhou. Findbugs: a program which uses static analysis to look for bugs in java code. In *Proceedings of the 2008 Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 1–4. ACM, 2010.

[9] Michael D Ernst, Marius B"uchi, Hermann H"artig, and Wolfgang Reif. Static enforcement of security with types. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 162–175. Springer, 2001.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. In *Proceedings of the Symposium on Semantics of Algorithmic Languages*, pages 1–38. ACM, 1969.

[11] Krzysztof R Apt and Nissim Francez. Ten years of hoare's logic: A survey–part i. In *Logics and Models of Concurrent Systems*, pages 11–52. Springer, 1981.

[12] Simão Melo de Sousa. Merciless bug hunting. why and how? 2015.

[13] Edmund M Clarke, Orna Grumberg, and Doron A Peled. Model checking. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):1305–1322, 1999.

[14] Tadao Murata. Petri nets: properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[15] Hadiqa Alamdar Bukhari and Sidra Sultana. Modeling and verification of car parking system. *ICSEA 2019*, page 235, 2019.

[16] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts, 10th Edition*. Wiley, 2018.

[17] Franco Cicirelli, Libero Nigro, and Francesco Pupo. Modelling and verification of concurrent programs using uppaal. *ECMS 2011*, 2011. https://www.scs-urope.net/conf/ecms2011/ecms2011